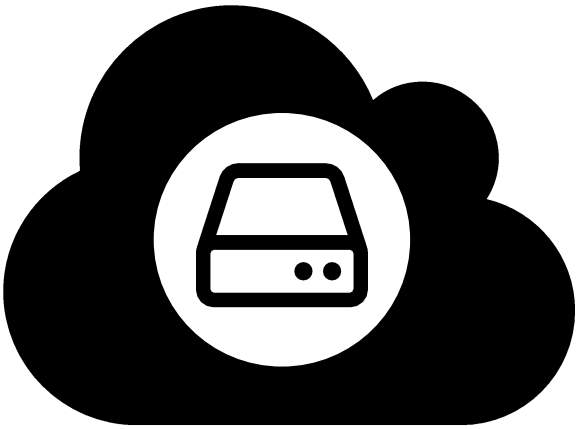


Sistema de Armazenamento de Ficheiros Hybrid Cloud

Gonçalo Manuel Duarte Lourenço
Mestrado Integrado de Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2014

Orientador
Sérgio Crisóstomo, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

Coorientador
Rui Prior, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

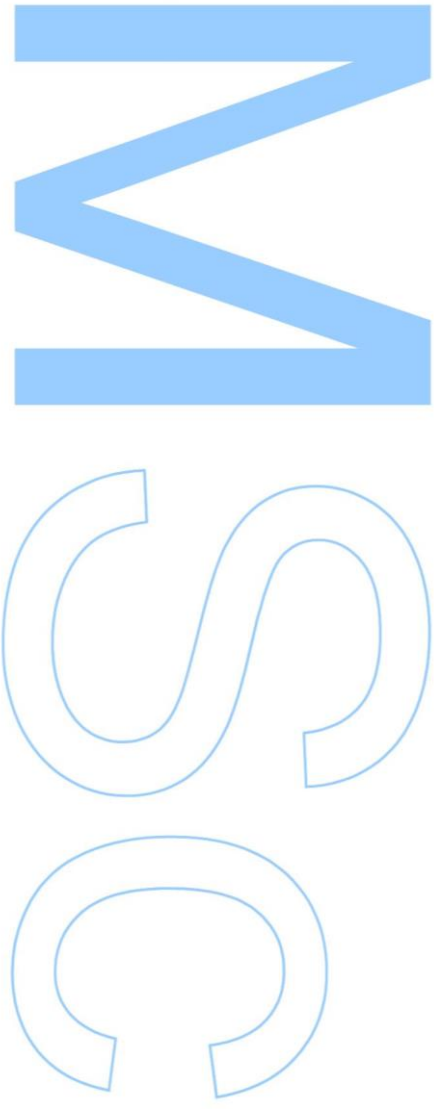
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Dedicado aos Srs. Drs. Profs. Rui Prior e Sérgio Crisóstomo pelo conhecimento e paciência infindáveis que demonstraram, ao Sr. Dr. Prof. Pedro Brandão por todo o suporte que amavelmente forneceu, aos meus pais e irmão e aos meus colegas com os quais convivo todos os dias.

Abstract

File storage is a universal necessity in computing systems. Concentrating storage needs in dedicated systems makes managing these needs easier, which in turn makes the existence of such systems essential in data centers. Recently, offshoring storage services using cloud systems has become a desirable option. Their on-demand functionality, ubiquity, and ease of use make these services very desirable. However, their usage presents some risks: storage security solutions offered by these services aren't always satisfactory, and possible availability problems can make these temporarily inaccessible to users.

This work focuses on the development of a hybrid file storage system using local resources and public cloud storage services. In this system, availability issues are mitigated through the usage of multiple cloud service providers in conjunction with redundancy mechanisms. Using cryptographic techniques, the system also offers confidentiality and integrity to stored data, even in the event of one of the providers becoming an opponent, be it due to internal, governmental, or external pressure, or having its structural security compromised.

Keywords — Clouds, storage, security, redundancy, availability

Resumo

O armazenamento de ficheiros é uma necessidade praticamente universal em sistemas computacionais. A centralização do armazenamento em sistemas dedicados torna a sua gestão mais fácil e flexível, pelo que a existência desses sistemas é essencial nos centros de dados. Recentemente, tem-se verificado uma tendência crescente para a deslocalização desse serviço para sistemas *cloud*. O funcionamento *on-demand*, a sua omnipresença e facilidade de uso tornam esta solução bastante atrativa. Porém, ela apresenta também alguns riscos: a segurança da informação armazenada nestes serviços nem sempre é satisfatória, e quebras de disponibilidade podem tornar o sistema temporariamente inacessível aos utilizadores.

Este trabalho consiste no desenvolvimento de um sistema de armazenamento de ficheiros híbrido, com recurso a discos locais e a serviços públicos de armazenamento *cloud*. Neste sistema, as quebras de disponibilidade são mitigadas através do uso de vários fornecedores diferentes de armazenamento *cloud*, combinado com mecanismos de redundância. Recorrendo a técnicas criptográficas, o sistema garante também a confidencialidade e integridade da informação armazenada, mesmo que um dos fornecedores de serviço de armazenamento *cloud* utilizados se torne um adversário, por motivos próprios, pressão de um governo ou outra entidade externa, ou violação da sua própria segurança. O sistema foi implementado no sistema operativo GNU/Linux.

Palavras-chave — *Clouds*, armazenamento, segurança, redundância, disponibilidade.

Conteúdo

Lista de Figuras

Acrónimos

ACL Access Control List

AES Advanced Encryption Standard

API Application Programming Interface

AWS Amazon Web Services

BUSE Block Device in User Space

CBC Cipher Block Chaining

CDMI Cloud Data Management Interface

CIMI Cloud Infrastructure Management Interface

CSP Cloud Service Provider

ESSIV Encrypted Salt-Sector Initialization Vector

FUSE File System in User Space

GCC GNU Compiler Collection

GCM Galois Counter Mode

GPL GNU Public License

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IV Initialization Vector

LRU Least Recently Used

LUKS Linux Unified Key Setup

MAC Message Authentication Code

MIT Massachusetts Institute of Technology

NBD Network Block Device

NIST National Institute of Standards and Technology

PaaS Platform as a Service

POSIX Portable Operating System Interface

RAID Redundant Array of Independent Disks

REST Representational State Transfer

SaaS Software as a Service

SHA Secure Hash Algorithm

SOAP Simple Object Access Protocol

SSD Solid State Drive

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TRIM Trim Command

URL Uniform Resource Locator

XML Extensible Markup Language

XTS XEX-based Tweaked-codebook mode with ciphertext Stealing

Capítulo 1

Introdução

Armazenamento de informação em recursos computacionais é essencial para a utilização destes. Porém, o espaço disponível nem sempre é suficiente. Para a satisfação desta necessidade universal, várias soluções foram desenvolvidas, algumas recorrendo a infraestruturas criadas pelos seus utilizadores, outras baseadas na utilização de espaço remoto cedido por terceiros em troca de alguma compensação (geralmente monetária).

O primeiro tipo de solução baseia-se no modelo convencional de computação [1], cujo funcionamento está dependente de infraestruturas físicas, locais e centralizadas de nome centro de dados (*data centers*). Estes centros podem incorporar um leque vasto de recursos, desde *hardware* a *software*. Para a constituição de uma solução no âmbito deste modelo e, por conseguinte, a criação de um centro de dados, é necessário um investimento prolongado e escrupuloso se se desejar manter os custos o mais reduzidos e controlados possível. A manutenção deste tipo de infraestrutura pode ser também cara, sendo difícil atingir um equilíbrio entre qualidade e custos.

Graças à proliferação de acesso rápido e barato à Internet, surgiu um novo modelo de computação baseado na utilização de recursos remotos. A este modelo foi dado o nome de computação *cloud* [2]. Inserido no âmbito deste modelo encontram-se vários recursos, todos eles possuidores de um funcionamento *on-demand*, podendo ser escalados em função dos seus custos e das necessidades dos seus utilizadores, e de uma existência omnipresente, desde que haja uma ligação à Internet. A aparência constante de recursos computacionais infinitos com custos dinamicamente escaláveis tornaram este modelo num elemento crucial para as necessidades atuais.

A apresentação destes recursos a utilizadores é realizada como um serviço, ao qual se dá o nome de serviço *cloud*, podendo cada recurso ser apresentado como um serviço diferente e único. Numa tentativa de saciar a necessidade de espaço de armazenamento, um dos serviços *cloud* consiste na apresentação de espaço de armazenamento. A este serviço é dado o nome de serviço de armazenamento *cloud*. À entidade que disponibiliza estes serviços é dado o nome de Cloud Service Provider (CSP), sendo o acesso aos serviços desta entidade feito através de um único ponto de entrada, mas recorrendo a diversas interfaces com diferentes graus de abstração. Ao conjunto de *hardware* e *software* da infraestrutura do CSP é dado o nome de *cloud* [1].

Apesar das vantagens que estes serviços oferecem, existe um conjunto de desafios que os utilizadores enfrentam ao os usarem [3]. Este trabalho tenta enfrentar dois dos principais desafios que decorrem da utilização de serviços de armazenamento *cloud*: segurança e disponibilidade.

1.1 Desafios

Nos serviços de armazenamento *cloud*, destacam-se os seguintes desafios de segurança [3][4]:

1. **Acesso Privilegiado** Dependendo do CSP que disponibiliza o serviço *cloud* de armazenamento, pessoas com diferentes graus de confiança podem ter acesso privilegiado aos dados armazenados. O utilizador pode não ter a possibilidade de especificar quem pode ou não ter este acesso na infraestrutura física, pois tal responsabilidade está a cargo do CSP que contratou;

2. **Segregação de Dados** Um serviço *cloud* pode não pertencer exclusivamente a um utilizador. Como tal, os dados destes podem estar misturados na infraestrutura de um CSP sem que haja conhecimento disso;

3. **Proteção de Dados** Os dados de um utilizador mantidos num CSP não são necessariamente cifrados. Quando são, um utilizador não sabe quem tem acesso às chaves de cifragem. Independentemente de serem ou não cifrados, existe ainda a questão de como um CSP procede à eliminação de dados ou à remoção de dispositivos de blocos;

Para além destes problemas de segurança observados por parte de utilizadores, existem ainda outras questões do lado do CSP. Uma entidade destas tem à sua responsabilidade uma infraestrutura vasta, na qual vários problemas de segurança podem surgir, uma situação

diretamente relacionada com a questão de segurança de centros de dados [5]. [4] indica que segurança deve ser garantida pelos CSPs a nível de servidores, bases de dados e rede interna (incluindo acesso à Internet).

Para a proteção de dados dentro das infraestruturas, recorre-se à utilização de técnicas de criptografia. Estas podem-se exercer sobre ficheiros individuais ou dispositivos de blocos inteiros. Do uso pleno e correto destas técnicas, surgem desafios influenciados pelos utilizadores, que acedem aos seus dados de forma exclusivamente remota, e pelos próprios CSPs, que lidam com estes dados no interior dos seus centros de dados. Estes desafios podem ser resumidos da seguinte forma:

1. **Confidencialidade** Os dados armazenados num CSP devem ser mantidos confidenciais, de forma a que agentes mal-intencionados exteriores (Internet) e interiores (trabalhadores do CSP) não os possam aceder;

2. **Velocidade** Operações de armazenamento e obtenção de dados devem ser rápidas, independentemente de onde na infraestrutura do CSP estes são armazenados. Isto deverá ser válido tanto a nível micro (interior de um centro de dados), como a nível macro (geo-localidade contratada). Para além da latência do acesso a um dispositivo de blocos, existe ainda a latência inerente ao acesso remoto, tornando este elemento crucial;

3. **Integridade** Devem existir mecanismos que garantam que uma mensagem não foi alterada por um agente interno ou externo ao CSP do utilizador, ou seja, devem existir mecanismos que garantam a integridade dos dados;

4. **Tamanho** O método de cifragem não deve desperdiçar espaço de disco. Como os serviços *cloud* funcionam numa base *on-demand*, é importante para o utilizador que esta condição se verifique, pois quanto mais espaço ocupar, mais caro o serviço sairá ao utilizador, tanto em termos de armazenamento (mais espaço ocupado significa mais espaço para o qual se terá de pagar) como em termos de transferência de dados (quanto maior o volume de dados, maior será a quantidade de dados a transferir, transferências essas que podem ser cobradas ao gigabyte).

Para além da questão de segurança e privacidade, [3] revela ainda que os serviços *cloud* podem também apresentar desafios no foro da disponibilidade. O impacto da impossibilidade de acesso pode ser significativo e, como tal, é desejável evitar este tipo de situação. [6] deixa claro

que estes desafios são reais e podem ter origem em vários fatores diferentes, como descrito em [7]. Estas falhas podem ter um período variado, mas possivelmente prolongado, estando estas diretamente relacionadas com o impacto que podem produzir junto dos utilizadores. [8] e [4] indicam que os principais pontos desta problemática são:

1. **Falha Centralizada** Visto que uma *cloud* tem um ponto centralizado de acesso, as consequências de uma falha desse mesmo ponto podem ser profundas no que toca a acessibilidade;
2. **Localidade de Dados** Um utilizador pode ter dificuldades em saber onde ao certo se localizam os seus dados. Tal pode-se dever ao nível de abstração que lhe é fornecido aquando do decorrer de um acesso, da quantidade de redundância oferecida, da geo-localidade (ou falta dela) ou ainda ao facto de as *clouds* terem um único ponto centralizado de acesso;
3. **Recuperação de Dados** Dependendo do serviço contratado por um utilizador ou do CSP escolhido por este, os seus dados podem ter um grau variável de replicação. Se tal grau se revelar insuficiente no momento em que se é necessário recorrer a este sistema, pode ocorrer perda de dados;
4. **Abstração de CSP** Ao contrário de outros modelos de computação, não existe um esforço conjunto para a criação de padrões de interoperabilidade entre diferentes CSPs. Isto torna um processo de migração complexo e caro, o que pode levar a processos de migração prolongados, durante os quais vários dados podem ficar indisponíveis.

Para mitigar os efeitos dos problemas de disponibilidade, pode-se recorrer a redundância. A ideia é manter uma ou mais cópias de dados armazenados nos serviços *clouds* de forma a que, se uma *cloud* estiver indisponível, seja possível na mesma acedê-los. Esta redundância pode ser atingida recorrendo a vários tipos de técnicas, cada uma com diferentes características de gasto de espaço e de tráfego.

Redundância implica mais do que manter dados disponíveis durante momentos de falha de disponibilidade, esta pode ser uma ferramenta essencial na segurança também. Espalhar os dados por diferentes CSPs pode ser crucial em garantir que um dos CSPs contratados pelo utilizador não tem acesso completo aos seus dados. Para além do mais, as técnicas utilizadas para garantir essa redundância podem ofuscar os dados. É também uma forma de evitar que um utilizador fique preso a um só CSP, pois, para atingir essa redundância, pode-se recorrer a diferentes CSPs em simultâneo.

1.2 Motivação e Objetivos

Os problemas de segurança e de disponibilidade no que toca a armazenamento de dados são uma realidade deste modelo de computação. Como [3] deixa claro, a utilização deste modelo traz várias vantagens para os seus utilizadores. Porém, para a sua plena utilização, utilizadores necessitam de ter em conta todos estes problemas, arriscando-se a lidar com diversas consequências caso não o façam. A resolução destes nem sempre é trivial, especialmente quando os CSPs não colaboram devidamente de forma a solucioná-los. A situação piora quando um utilizador recorre a múltiplos CSPs ao mesmo tempo para satisfazer as suas necessidades, pois este poderá ter que acordar termos de uso específicos em cada caso. Perante CSPs indisponíveis a cooperarem, é necessário o utilizador tomar a iniciativa.

O objetivo deste trabalho é a descrição de um sistema de armazenamento de ficheiros com recurso a discos locais e a serviços públicos de armazenamento *cloud*. O sistema tem em conta questões de privacidade (a informação armazenada nos serviços de *cloud* públicos deve permanecer confidencial) e de disponibilidade (a indisponibilidade de um dos serviços de *cloud* públicos não dever afetar a disponibilidade do nosso serviço). Através deste sistema, pretende-se fornecer a utilizadores de serviços *cloud* uma ferramenta capaz de lidar com todas estas questões de forma competente e eficiente, melhorando, desta forma, as suas experiências com este tipo de modelo de computação.

Este documento encontra-se dividido em diversos capítulos. Capítulo 2 introduz os conceitos-base necessários para a compreensão da maior parte das tecnologias utilizadas pelo sistema; capítulo 3 dá uma visão de alto-nível para cada um dos componentes lógicos do sistema; capítulo 4 descreve uma implementação do sistema, realizada no sistema-operativo GNU/Linux; capítulo 5 expõe várias questões práticas que surgiram durante o desenvolvimento da implementação e do sistema.

Capítulo 2

Trabalho Relacionado

No âmbito do nosso trabalho, foram estudadas as principais características que definem os serviços de armazenamento *cloud*. De forma a uniformizar o acesso a múltiplos serviços *cloud*, foram estudadas diferentes soluções de abstração. Para tornar o acesso a estes serviços local, foram estudadas formas de mapeamento de funções dos serviços *cloud* em funções de sistemas locais. O objetivo de obtenção de segurança no uso destes serviços levou ao estudo de soluções criptográficas. A questão de disponibilidade é tratada através do estudo de soluções de obtenção de redundância.

2.1 Soluções de Armazenamento Cloud

Nesta secção, é descrita a taxonomia das *clouds*, de forma a ser possível identificar os diferentes serviços disponibilizados por CSPs. De seguida, é descrito a organização dos serviços de armazenamento. Finalmente, são abordados os aspetos de acesso a estes serviços e as opções disponibilizadas para proteção de dados.

2.1.1 Taxonomia das Clouds

Um serviço *cloud* é disponibilizado por um CSP. A integração de um serviço com a infraestrutura de um utilizador pode variar de caso para caso, dependendo da proximidade existente entre este e o CSPs que contratou [3] [6]. Dependendo deste grau, uma dada *cloud* pode ser classificada como sendo de um dos seguintes quatro tipos:

- **Privada** Uma *cloud* deste tipo está altamente relacionada com o utilizador. A sua infraestrutura pode pertencer ao utilizador, assim como a sua manutenção, o que significa que está completamente adaptada às suas necessidades. Somente o utilizador pode tomar proveito desta *cloud*;
- **Comunitária** *Clouds* designadas comunitárias são somente *clouds* privadas nas quais múltiplos utilizadores participam de forma cooperativa;
- **Pública** O tipo mais tradicional de *cloud*. Disponibilizada ao público em geral por uma entidade separada do utilizador, ficando a sua gestão a cargo dessa entidade;
- **Híbrida** Como o próprio nome implica, este tipo envolve a mistura de *clouds* públicas com privadas/comunitárias.

Uma *cloud* privada naturalmente tem tempos de acesso diferentes das de uma pública, pois estas podem ser integradas diretamente na infraestrutura do utilizador. Porém, a capacidade de aprovisionamento de recursos computacionais de uma *cloud* privada não será necessariamente igual ao de uma pública. Uma *cloud* híbrida é uma tentativa de equilibrar a personalização de uma *cloud* privada com as restrições impostas por um CSP numa *cloud* pública.

Existe ainda um outro sistema de classificação quando se pretende descrever uma *cloud*, dependente do tipo de serviço que presta. Este serviço não precisa de estar diretamente relacionado com somente um recurso computacional (p.ex. espaço,ou processamento), podendo ser constituído por mais que um recurso, inclusive *software*. Dependendo do que é fornecido, o serviço prestado por um CSP pode ser classificado de três formas diferentes [3] [6]:

- Software as a Service (SaaS) consiste na disponibilização de programas de computador aos utilizadores, cobrindo apenas o tempo durante o qual estes são utilizados. Ao contratar este tipo de serviço, um utilizador não precisa de se preocupar com atualizações aos programas ou o uso de *hardware* específico para os executar, acabando por se perder dessa forma a capacidade de executar versões específicas e a possibilidade de controlar diretamente o que está ao certo a ser executado. Um exemplo deste tipo de serviço é o Google Apps;
- Platform as a Service (PaaS) revela um ambiente de alto-nível para desenvolver, testar e lançar aplicações *online*. A manutenção do ambiente em si fica a cargo do CSP, aliviando o utilizador de tal administração. Isto também significa que, devido à possível falta de estado do ambiente, torna-se impossível modificá-lo de acordo com as necessidades do utilizador.

Exemplos deste tipo de serviço são o Google App Engine, o Microsoft Azure e a Amazon Web Services (AWS);

- Infrastructure as a Service (IaaS) refere-se ao fornecimento direto de recursos computacionais para um controlo total destes por parte do utilizador. Isto implica que a administração destes recursos fica inteiramente a cargo do utilizador, o que pode ser constituído como um problema.

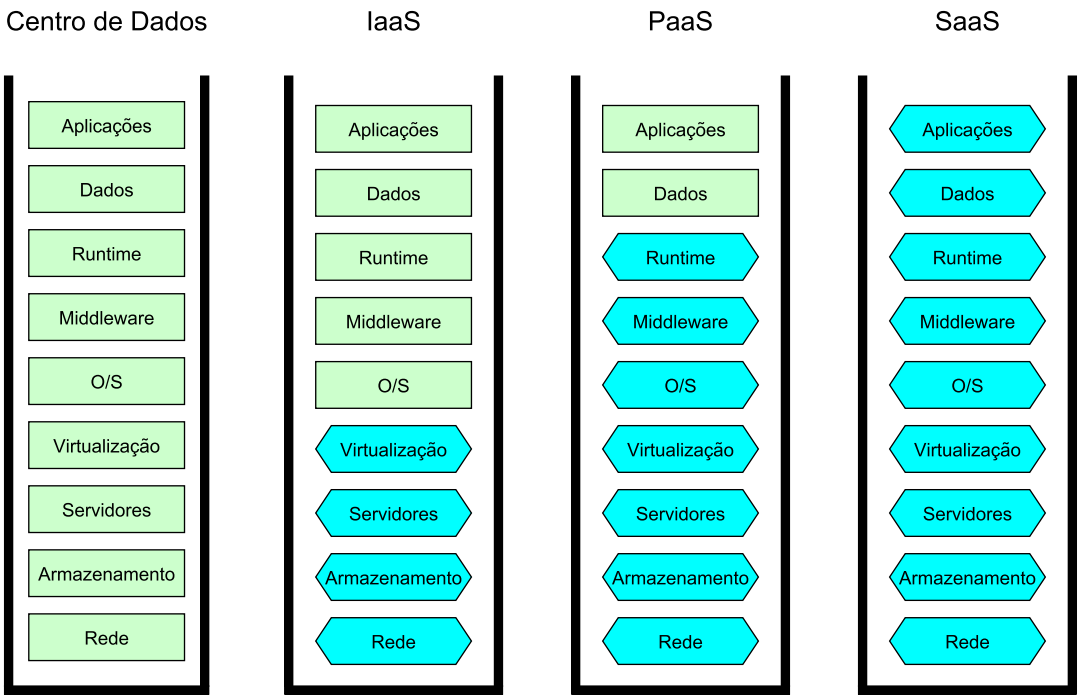


Figura 2.1: Diagrama de Serviços *Cloud*

Diagrama que representa os diferentes graus de serviços disponibilizáveis por um CSP. Quadrados indicam elementos geridos pelo utilizador, hexágonos são geridos por CSPs.

2.1.2 Serviços de Armazenamento de Dados

A disponibilização de espaço para armazenamento de dados só é revelado ao utilizador se este contratar um serviço do tipo PaaS ou IaaS. A contabilização pode ser realizada sobre o espaço que o utilizador gasta, por conjuntos de um determinado número de operações sobre o espaço, por tipo de redundância ou ainda por geo-localização [9] [10].

O funcionamento de um serviço de armazenamento *cloud* é semelhante ao de um sistema de ficheiros local. É possível realizar a seguinte analogia: este tipo de serviço permite armazenar ficheiros em pastas num espaço ilimitado de armazenamento, espaço esse que é alocado dinamicamente. A um ficheiro é dado o nome de *blob* ou objeto, enquanto uma pasta é designada por balde (*bucket*) ou contentor (*container*) [11] [12] [13]. O acesso a este espaço é realizado diretamente, sem ser necessário recorrer a um outro serviço intermediário para o fazer.

Ao contrário dos sistemas de ficheiros em que uma pasta pode ter sub-pastas, um contentor não pode ter dentro de si outros contentores. Isto significa que, para guardar uma pasta na *cloud* com sub-pastas, é necessário criar um contentor para a pasta principal e para cada sub-pasta.

Um ficheiro é diretamente equiparável a um objeto. Para armazenar um ficheiro no serviço, não é necessário modificá-lo de forma alguma. Cada objeto possui um conjunto de metadados, podendo estes ser acedidos separadamente do ficheiro. No serviço Amazon S3 [14], os metadados incluem o tamanho do objeto, data de última modificação e de criação e ainda um valor MD5, sendo ainda possível criar campos adicionais pelo utilizador.

É possível ainda recorrer a serviços de computação *cloud* para obter espaço de armazenamento igual ao de um dispositivo de blocos. O funcionamento destes está dependente do uso de máquinas contratadas no âmbito do serviço de computação, o que implica a contratação de dois serviços em simultâneo.

2.1.3 Acessibilidade das Clouds

A interação de um utilizador com um serviço *cloud* que tenha contratado é regulada pelas opções fornecidas pelo CSP desse mesmo serviço. No que toca a interações programáticas, um número significativo de CSPs usam soluções (Application Programming Interfaces (APIs)) proprietárias, o que torna necessário escrever código específico para cada CSP utilizado. Esta problemática torna a transição entre diferentes CSPs uma operação cara e demorada [3].

Para enfrentar este desafio de falta de acesso uniforme nos serviços de armazenamento, foi criado um padrão chamado Cloud Data Management Interface (CDMI) [13]. Este define funções para todas as operações típicas de manipulação de dados, como inserção, eliminação, leitura e escrita. O uso deste padrão é reduzido; as únicas utilizações deste encontram-se enumeradas em [15].

Em vez de usarem este padrão, diversos CSPs preferem implementar as soluções dos seus principais concorrentes no mercado. Um exemplo disso é a interface AWS, que proliferou devido à popularidade de que o serviço goza atualmente [16]. Desta forma, CSPs podem cativar os utilizadores deste serviço popular a aderirem aos seus próprios.

Independentemente de qual API é necessária usar, a troca de mensagens entre utilizador e CSP é realizada através de pedidos Representational State Transfer (REST) ou Simple Object Access Protocol (SOAP) [17]. O primeiro consiste na troca de mensagens Hypertext Transfer Protocol (HTTP), enquanto o segundo baseia-se no envio e receção de mensagens Extensible Markup Language (XML).

2.1.4 Opções de Proteção de Dados

Utilizadores podem ter opções à sua disposição para proteger os seus dados. A disponibilização destes depende do CSP do serviço contratado. Uma destas opções é a utilização de Access Control Lists (ACLs), listas que consistem na especificação das relações de utilizadores de um dado espaço de armazenamento com cada objeto contido nesse espaço. Através destas listas, é possível descrever o que é que um dado utilizador pode ou não fazer a um objeto. ACLs podem ser relativas a objetos individuais ou a contentores. Outras tecnologias de controlo de acesso podem ser utilizadas, dependendo do CSP utilizado. O serviço AWS é um exemplo do uso de ACLs [18].

Para além de controlo de acesso, é também possível recorrer a mecanismos de cifragem de dados, disponibilizados nas APIs dos CSPs. A gestão das chaves de cifragem e a execução do processo de cifragem em si podem ficar a cargo dos CSPs ou do utilizador, dependendo do funcionamento da APIs. No caso em que a gestão das chaves fica a cargo dos CSPs, há certas questões de segurança de relevo. Uma delas é a questão de acesso privilegiado: nem sempre é possível ao utilizador saber quem gere elementos tão cruciais para a correta cifragem dos dados. [19] descreve o caso do AWS.

Também relevante é a questão do transporte desses mesmos dados de forma segura até ao seu destinatário, quer tenham origem num CSP ou num utilizador. Para tal, os CSPs recorrem ao uso da tecnologia Secure Sockets Layer (SSL), o que permite cifrar os dados em transporte pela Internet.

Existem outros pormenores de segurança mais específicos ao funcionamento interno de cada CSP que se podem encontrar explicitados nos contratos celebrados entre estes e os utilizadores. [4] analisa possíveis riscos de segurança que são relevantes aos utilizadores, mas cuja a resolução e funcionamento variam muito entre os diversos CSPs.

2.2 Abstração de Acesso

O acesso às *clouds* não é trivial. [16] indica que a escolha de um CSP implica também a escolha de uma API em específico. A aposta numa API implica um investimento monetário e temporal significativo que pode ser potencialmente desperdiçada se o utilizador, a certa altura, decidir trocar de CSP. Ficar dependente de uma API é, portanto, contraproducente e indesejável, o que cria a necessidade de alguma camada de abstração no acesso a CSPs.

Coloca-se, então, a questão de como um utilizador pode evitar esta situação penosa [20]. Em vez de recorrer a uma API proprietária, pode-se utilizar um padrão aberto e esperar que eventualmente seja adotado. Porém, a proliferação destes padrões é reduzida, o que pode causar transtornos que não surgiriam com o uso das APIs proprietárias. Em vez disso, pode-se tentar estruturar um programa em camadas, de forma a que seja possível dividi-lo numa camada lógica, na qual o funcionamento do programa em si está descrito, e numa camada *cloud*, onde a interação com as *clouds* é realizada.

Para fazer frente a esta necessidade criada pela falta do uso de padrões e para tentar facilitar a estruturação de programas em camadas, surgiram ferramentas encarregues de implementar uma camada *cloud* para os programas. A estas ferramentas foi dado o nome de camada de abstração. Estas podem-se dividir em duas principais categorias: opções dependentes de uma linguagem de programação e opções independentes destas.

O funcionamento destas ferramentas está baseado na implementação de funções genéricas que podem ser traduzidas para funções específicas da API de cada CSP [20]. O processo de tradução é realizado por componentes chamados *drivers*, que funcionam de forma modular, permitindo desenvolver novos componentes deste tipo sem perturbar a estrutura geral do programa [21]. A figura 2.2 ilustra este funcionamento para opções dependentes e independentes de linguagens de programação.

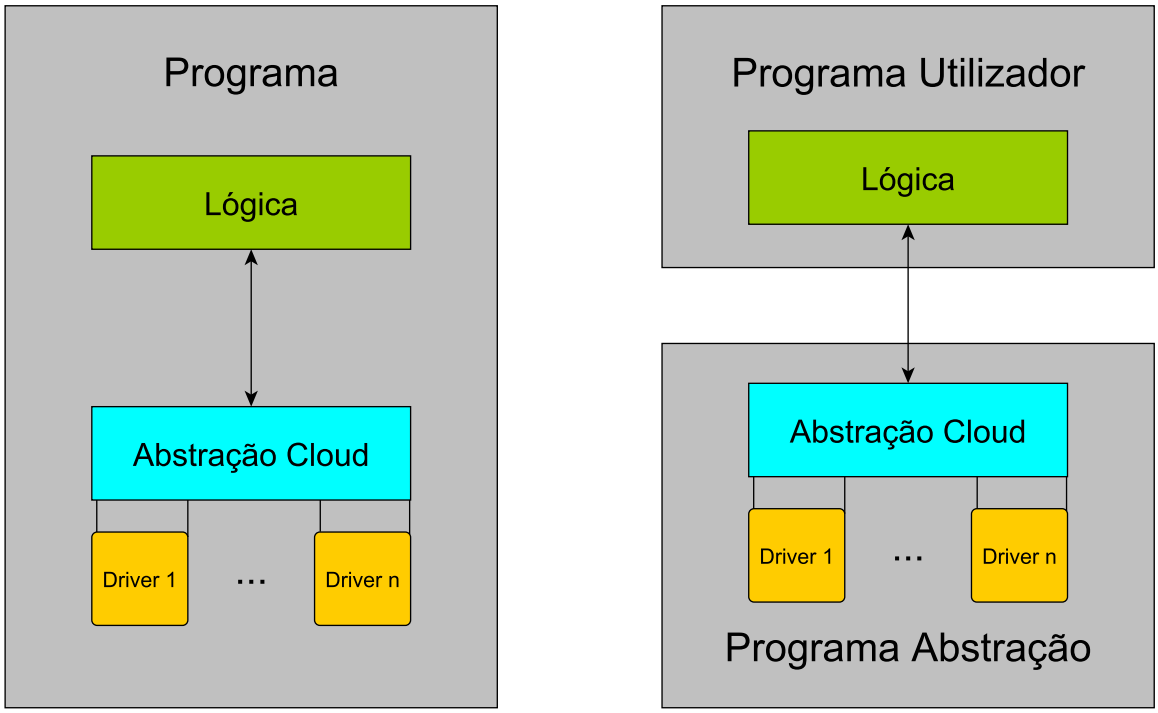


Figura 2.2: Tipos de Ferramentas de Abstração

Diagrama que representa a estrutura dos dois tipos diferentes de ferramentas de abstração. Do lado esquerdo, está representado um programa que utiliza uma ferramenta específica de linguagem; do lado direito, encontra-se descrito o funcionamento de um programa que usa uma ferramenta independente de linguagem. No caso da direita, a ferramenta é um programa isolado do do utilizador.

Todas as soluções em ambas as opções estão limitadas no número de CSPs que suportam, assim como no número de funções que podem implementar. A limitação nos CSPs suportados depende principalmente a maturidade destes projetos. Visto que um elevado número destes projetos são *open-source*, o desenvolvimento de novas *drivers* pode ser realizado pela comunidade, evitando-se dessa forma a implementação de critérios de escolha no que toca a quais CSPs suportar. Quanto ao número de funções, a razão pela qual existe um limite deve-se ao facto de certas funções fornecidas por CSPs serem demasiado específicas à plataforma destes, sendo impossível traduzi-las para funções genéricas.

A utilização deste tipo de ferramenta é útil somente se o utilizador pretender ter implementações ágeis, capazes de transitar entre diferentes CSPs sem grandes transtornos. Se um utilizador se quiser dedicar exclusivamente à plataforma de um dado CSP, então o uso de abstrações de acesso pode significar a perda de algumas funcionalidades que podem ou não ser desejáveis, dependendo do uso que o utilizador pretende dar.

No que toca a opções específicas a linguagens, existem os seguintes projetos que se apresentam como soluções para este desafio: Dasein Cloud [22], jClouds [23], CloudLoop [24], Zend Framework [25], Fog [26], pkgcloud [27], elibcloud [28] e Libcloud [29]. Dasein Cloud, jClouds e CloudLoop foram desenvolvidas para a linguagem Java; Zend Framework lida com a linguagem PHP; Libcloud funciona com Python; Fog é utilizado em Ruby; pkgcloud pertence a node.js; elibcloud é tratado em Erlang. Destas opções, somente o CloudLoop não se encontra em desenvolvimento. Todas estas opções apresentam-se como bibliotecas que podem ser utilizadas como substitutos às APIs fornecidas pelos CSPs contratados pelo utilizador.

No caso de ferramentas independentes de linguagens, existe apenas uma solução chamada Deltacloud [30]. Esta consiste num programa isolado do do utilizador, não existindo qualquer tipo de integração de um no outro. Para que o programa do utilizador possa utilizar a ferramenta, ambos necessitam de estar a ser executados em simultâneo. O seu funcionamento é semelhante ao de um *proxy*, pois recebe pedidos através do uso de três APIs RESTful diferentes (API própria, Cloud Infrastructure Management Interface (CIMI) e AWS), pedidos esses enviados por Transmission Control Protocol (TCP) recorrendo a HTTP, e efetua as interações com os CSPs por parte do programa do utilizador [21].

A utilização de bibliotecas permite uma integração direta nos programas do utilizador, evitando a introdução de latências desnecessários, algo que se verifica no uso do Deltacloud, visto que acomunicação com o *proxy* realiza-se através de uma ligação TCP. A utilização da segunda opção requer ainda o uso de uma biblioteca para interagir com o *proxy*, fator que é minimizado pelo suporte oferecido por algumas das bibliotecas supramencionadas. Esta segunda opção apresenta-se como uma solução viável em ambientes de múltiplos utilizadores com múltiplos programas diferentes, devido à sua acessibilidade neutra.

A abstração através de meios programáticos não é a única forma de abstração, visto existirem atualmente soluções pré-criadas que permitem solucionar esta questão de forma mais orientada a utilizadores com escassos conhecimentos técnicos. A estas soluções é dado o nome de Mediadores de Serviços *Cloud* (*Cloud Service Brokers*). De acordo com [2], um mediador de *clouds* é um agente que interage com CSPs por parte de um utilizador, sendo capaz de realizar negociação, arbitragem e agregação, sendo este último o elemento relevante para a questão de abstração de acesso.

2.3 Tornar o Acesso Local

A interação com um serviço *cloud* é bidirecional. Torna-se necessário, então, gerir a forma como as operações do serviço *cloud* são mapeadas num sistema local, assim como a granularidade deste mapeamento.

A gestão pode ser realizada de três formas diferentes: recorrendo ao sistema de ficheiros local; implementando o nosso próprio sistema de ficheiros; criando o nosso próprio dispositivo de blocos. Enquanto a primeira opção pode ser independente do sistema-operativo no qual o programa do utilizador está a ser executado, as restantes duas opções não, visto lidarem com componentes intrínsecos ao funcionamento de um sistema-operativo. Cada uma destas três opções apresenta um diferente nível de granularidade, derivado do seu funcionamento. Nesta secção, a discussão sobre as últimas duas formas é realizada de uma perspetiva do sistema-operativo GNU/Linux.

A utilização do sistema de ficheiros local consiste no uso direto das funções do sistema-operativo para a manipulação de ficheiros. Os dados de cada ficheiro armazenado na *cloud* são mapeados localmente para um ficheiro. No entanto, os metadados mantidos pelo sistema de ficheiros local irão ser diferentes daqueles que estão no serviço *cloud*. Funções realizadas sobre uma *cloud* não são mapeadas diretamente a funções locais e vice-versa. Torna-se necessário o uso de um programa intermediário para realizar a tradução. Ou seja, o sistema local não terá conhecimento do espaço de armazenamento remoto.

Para mapear funções locais a funções remotas sem uso de intermediários, é necessário ter um controlo fino na forma como as operações locais são executadas pelo sistema-operativo. Visto que tais operações locais são descritas em módulos de kernel [31], uma possível solução é ter um módulo novo que implemente um sistema de ficheiros novo.

Criar um módulo destes não é trivial. Programação a nível de kernel de um qualquer sistema-operativo acarreta consigo um conjunto de razões que tornam a execução de tal tarefa indesejável [32][33]: a curva de aprendizagem é elevada devido à falta de ferramentas que facilitem a tarefa; a gestão de memória torna-se complexa pois, se não for bem realizada, pode ter consequências drásticas para o funcionamento corrente do sistema-operativo; obriga ao uso da linguagem de programação C, mas sem poder utilizar as bibliotecas padrão, recorrendo, como alternativa, a estruturas e funções próprias e exclusivas ao kernel; o código produzido torna-se intrinsecamente ligado às características de um dado sistema-operativo. Para evitar estes problemas, foi criada a tecnologia o File System in User Space (FUSE) [34] no sistema GNU/Linux.

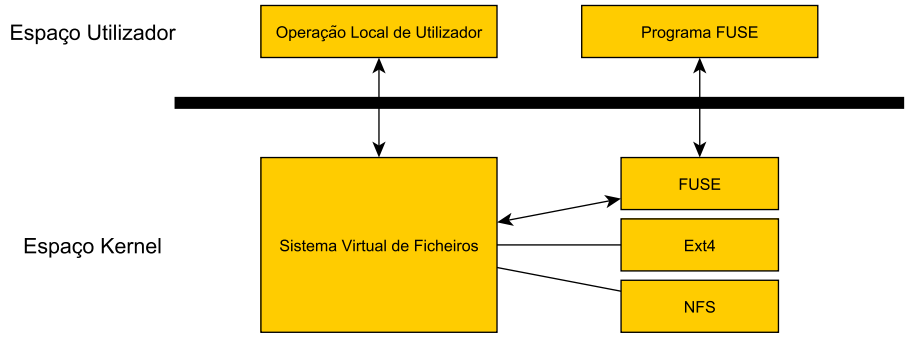


Figura 2.3: Funcionamento do FUSE

Diagrama que apresenta uma descrição simplificada do funcionamento do FUSE. O utilizador realiza uma operação que é passada do kernel até ao programa FUSE, graças ao módulo de kernel FUSE.

O FUSE permite programar um sistema de ficheiros fora do kernel. Esta tecnologia consiste na utilização de bibliotecas e de um módulo de kernel pré-programado para tornar isto possível. Um conjunto de funções é fornecido ao utilizador por implementar e este escolhe quais delas pretende desenvolver. Para além das funções, o utilizador pode introduzir informação adicional à execução do sistema de ficheiros, podendo torná-lo consciente da existência de espaço remoto. Existe um variado número de exemplos deste tipo de implementação que recorrem ao FUSE: s3fs [35] e azurefs [36] são dois deles. A figura 2.3 descreve a estrutura desta tecnologia.

Apesar da inexistência de intermediário poder ser vantajosa, o uso desta tecnologia implica já, por si só, uma certa abstração no que toca às operações realizadas sobre conteúdo na *cloud*. Um sistema de ficheiros não é capaz de aceder de forma "cega" dados guardados, isto é, para aceder a um conjunto de dados, tem de saber o que necessita de aceder. Nem sempre esta abstração é desejável: em certos casos, uma granularidade ainda mais fina pode ser desejável.

Se for esse o caso, uma solução é aceder ao nível de bytes/blocos individuais. Tal é possível através da utilização de dispositivos de blocos. Tal como no caso de sistemas de ficheiros, a forma como um dispositivo de blocos se comporta é descrito em módulos do kernel de um sistema-operativo. Estes módulos são *drivers* de dispositivos físicos. Porém, isso não impede a criação de uma *driver* que funcione com um dispositivo virtual, sem correspondência física.

Tal como no caso de sistemas de ficheiros, é desejável não ter que recorrer à programação a nível de kernel. Para tornar a tarefa de elaboração de um dispositivo de blocos virtual mais simples e para tornar estes dispositivos acessíveis remotamente, foi criado o Network Block Device (NBD) [37] no sistema GNU/Linux. Esta tecnologia permite criar um programa que

serve pedidos por bytes a clientes locais ou remotos. Nos clientes, existe um módulo kernel pré-programado que funciona como uma *driver* de dispositivo de blocos. O seu funcionamento encontra-se descrito de forma pormenorizada na secção 4.2 e ilustrado de forma resumida na figura 2.4.

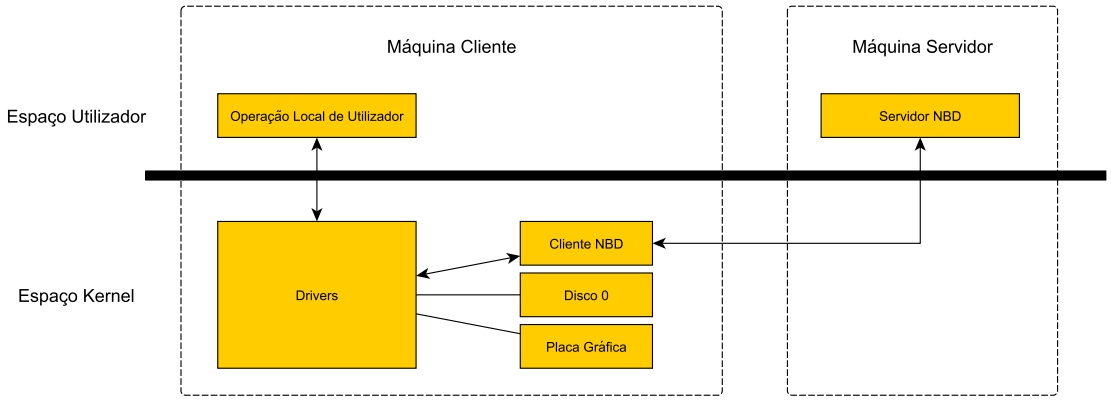


Figura 2.4: Funcionamento do NBD

Diagrama que descreve, de forma simplificada, o funcionamento do NBD. O utilizador realiza uma operação que é passada do kernel até ao servidor NBD, graças à *driver* existente no cliente NBD. Neste caso, o programa servidor está numa máquina separada, mas pode residir na mesma máquina que a do cliente.

Recorrendo a esta tecnologia, torna-se possível lidar diretamente com este grau de granularidade fino. Desta forma, podemos trabalhar com dados armazenados nas *clouds* de forma "cega". Como exemplo deste tipo de implementação, foi elaborado o projeto S3NBD [38]. De notar que certos elementos do NBD descritos na página do projeto já se encontram desatualizados.

2.4 Segurança de Armazenamento

Apesar de existirem mecanismos de controlo de acesso em serviços de armazenamento *cloud*, estes não são capazes de garantir confidencialidade dos dados. No âmbito da questão de segurança no armazenamento, torna-se necessário compreender conceitos-base e soluções de criptografia. O desenvolvimento desta questão está dependente da granularidade de acesso aos dados.

2.4.1 Conceitos de Criptografia

Para abordar esta questão de segurança, é necessário expor conceitos básicos da criptografia [39][5], área que se dedica à cifragem e decifragem de informação, isto é, que se dedica a manter mensagens seguras. Cifragem consiste no processo de disfarçar informação de tal forma que esconda o seu verdadeiro conteúdo; decifragem é o processo de desmascarar a informação disfarçada pelo processo de cifragem. Ambos os processos estão dependentes do uso de operações matemáticas que são exercidas sobre os dados originais. A estes processos matemáticos é dado o nome de cifra. Uma cifra garante apenas confidencialidade.

Para além de confidencialidade, a criptografia tem de ser capaz de responder a outros desafios:

- **Autenticação** Deve ser possível ao recetor determinar a origem dos dados; um agente mal-intencionado não deve ser capaz de se fazer passar por outrem;
- **Integridade** Deve ser possível ao recetor verificar se os dados foram ou não alterados desde o momento de transmissão deles; um agente mal-intencionado não deve ser capaz de substituir dados legítimos por ilegítimos;
- **Não-repudiação** Um transmissor não deve ser capaz de negar falsamente que a certo ponto enviou um conjunto de dados específico.

Todas as cifras modernas recorrem ao uso de um segredo chamado de chave para mascarar e desmascarar os dados. Funciona como uma senha secreta: se um utilizador a souber, tem acesso aos dados; caso contrário, o acesso é-lhe negado. Dependendo da cifra usada, uma ou mais chaves podem ser usadas. Às cifras que usam apenas uma chave são designadas de cifras simétricas; as que usam duas chaves têm o nome de cifras de chave pública.

Uma cifra simétrica é caracterizada por um conjunto de pormenores comum a todas. O processo de cifragem e decifragem estão dependentes do uso de uma chave. Sem esta, não se podem realizar. Transmissor e recetor devem saber de antemão a chave. A segurança deste tipo de cifra está totalmente dependente da chave: se esta for comprometida, então os dados também o são.

As cifras simétricas podem ter dois tipos de granularidade diferentes: bits/bytes individuais ou blocos de dados de tamanho fixo. Às primeiras é dado o nome de cifras de fio-de-água, enquanto as segundas são chamadas de cifras de bloco.

As cifras de fio-de-água são utilizadas em contextos em que se pretende cifrar uma entrada indefinida de dados. Dada uma chave, o mesmo bit/byte terá obrigatoriamente de ser cifrado para um bit/byte diferente. Para tal, estas cifras estão altamente dependentes de um componente chamado gerador *keystream*, que vai gerando conteúdo aleatório baseado na chave dada. Esse conteúdo é, então, usado para cifrar os dados. Para a cifragem ser segura, o estado inicial do conteúdo aleatório deve ser único, o que significa que deve ser usada uma chave diferente em cada invocação de uma cifra deste tipo. O uso deste tipo de cifra no contexto de armazenamento é inexistente.

Para as cifras de bloco lidarem com mais que o tamanho máximo que suportam, necessitam de um processo que encadeie os diferentes blocos de dados, de forma a que o resultado se mantenha seguro. A estes processos é dado o nome de modo de operação. Alguns destes requerem mais *input* para além da chave a se utilizar com a cifra, nomeadamente Initialization Vectors (IVs), que servem para tornar a relação entre diferentes blocos de dados impercetível. Por vezes, o termo IV é substituído por *nonce*. Porém, estes não são equivalentes: enquanto um IV pode ser derivado de forma previsível (como, por exemplo, un número de sequência), sendo somente necessário garantir que os valores sejam únicos, um *nonce* necessita de ser proveniente de uma fonte capaz de o gerar de forma aleatória e só pode ser usado uma única vez independentemente da chave utilizada.

Nas cifras de chave pública, são usadas duas chaves diferentes: uma chave, a chamada privada, é posse exclusiva do transmissor; a outra, chamada pública, pode ser revelada junto de qualquer recetor. O que se pretende com este tipo de cifra é que a chave usada para cifrar seja diferente da usada para decifrar. Recorrendo a este tipo de cifras, a troca de chaves simétricas torna-se trivial e segura. Porém, este tipo de cifra não é normalmente usada para cifrar grandes conjuntos de dados, pois o desempenho delas torna esse tipo de uso proibitivo.

De certa forma, a criptografia consiste na compressão de segredos: um segredo grande, neste caso um conjunto de dados, é comprimido para um pequeno segredo, uma ou mais chaves. É natural, então, que existam funções que tomem este conceito de compressão ao limite. A estas funções é dado o nome de funções de *hash*. O que estas funções fazem é receber um conjunto de dados e truncá-lo para um dado tamanho, de tal forma que a relação entre os dados originais e os dados truncados seja impercetível. Estas funções são irreversíveis, sendo impossível descomprimir dos dados depois de truncados, mas a função destas não é a transmissão de dados.

Funções de *hash* têm como propósito apresentar resumos criptograficamente seguros de um conjunto de dados. Estes resumos são chamados de *hashes* (*hash* no singular). Isto torna estas funções passíveis de serem utilizadas para a verificação de integridade de dados, isto porque representam, de forma inequívoca, o estado desse conjunto de dados aquando da produção do valor *hash*. Se se enviar um conjunto de dados, cifrados ou decifrados, e depois se enviar o valor de *hash*, pode-se alimentar os dados recebidos à mesma função de *hash* utilizada para calcular o *hash* recebido. Se o valor novo calculado for igual ao recebido, então a mensagem não foi alterada.

É possível integrar nestas funções a utilização de uma chave no processo de cálculo de *hashes* de tal forma que se obtenha a propriedade de autenticidade, pois um valor de *hash* resultante de tal processo só seria possível de produzir com uma certa chave a que somente um ou mais indivíduos têm acesso. As funções de *hash* que usam uma chave têm o nome de funções de *hash* criptográficas e produzem como resultado valores Message Authentication Codes (MACs).

Uma função de *hash* criptográfica deve ser possuir as seguintes resistências:

- **Resistente a Colisões** Encontrar dois conjuntos de dados para as quais o valor gerado é o mesmo;

- **Resistente a Pré-imagens** Dado um valor de MAC, tentar encontrar um outro conjunto de dados para qual o valor calculado seja o mesmo;

- **Resistente a Segunda Pré-imagem** Dado um conjunto de dados, encontrar um outro para qual o valor calculado seja o mesmo.

Um valor de *hash* resume o conteúdo de um conjunto de dados. Se for possível quebrar a relação entre o conjunto de dados original e o valor de *hash*, então torna-se possível quebrar uma das três condições supramencionadas. Se se realizar o cálculo do valor MAC com dados decifrados, pode-se estar a fornecer desnecessariamente informação sobre o conteúdo do conjunto de dados. Tal procedimento é desnecessário, pois se se proceder à cifragem antes do cálculo, a relação entre o conteúdo do conjunto de dados e o valor MAC produzido fica mais difícil de determinar. Logo, um valor MAC deve ser calculado a partir de dados cifrados.

2.4.2 Problemas com Armazenamento

Do uso de técnicas de criptografia surgem quatro principais desafios: confidencialidade, velocidade, tamanho e integridade. Na questão de segurança no armazenamento de dados, é necessário obter o primeiro ponto tendo em conta requisitos específicos, próprios desta questão, para a resolução dos restantes. A cifra a usar para obter confidencialidade terá de ser rápida (capaz de aguentar operações em tempo real), trabalhará com conjuntos de dados de tamanho previsível (blocos ou ficheiros), o seu resultado não pode ocupar mais espaço que os dados originais e, por conseguinte, não tem de ser capaz de garantir integridade.

As características destes processos estão relacionadas com a escolha do tipo de cifras a utilizar. Cifras de chave pública possuem um desempenho penoso que tornam o seu uso para a cifragem de grandes volumes de dados proibitivo. O uso de cifras de fio-de-água seria impraticável, pois como é necessário ter uma chave diferente para cada invocação, seria necessário manter uma lista de associações entre chaves e estado dos dados para cada conjunto de dados. Sendo assim, deve-se utilizar cifras simétricas de bloco. Isso significa o uso de uma só chave para cifrar e decifrar um conjunto de dados.

Perante o uso de cifras de chave simétrica e a possibilidade de lidar com conjuntos de dados de tamanho maior que o *input* de uma cifra deste tipo, torna-se necessário escolher um modo de operação. Esta escolha é determinante para garantir a condição de espaço gasto, pois certos modos de operação levam à produção de valores maiores que os dados a serem cifrados. Esta condição deriva da utilização de dispositivos de blocos físicos, em que a unidade-base, o bloco, tem um tamanho fixo e imutável.

A obtenção de integridade deveria ser realizada através da utilização de valores MAC. Existem modos de operação que produzem estes valores ao mesmo tempo que cifram dados. Porém, devido à restrição de espaço, a utilização de valores MAC não se pode realizar, pois é impossível garantir que, para todas as unidades de dados, a cifragem resultará num objeto de tamanho reduzido suficiente que permita encaixar na unidade o valor de MAC correspondente. Sendo assim, se a condição de gasto de espaço tiver de ser exercida, não é possível nem garantir integridade nem usar os modos de operação mencionados.

Para além destas garantias, uma solução para o problema de armazenamento de dados deve ser capaz de:

- Cifrar dados sem os perder durante o processo, pois se tal se verificar, os dados são perdidos de forma permanente;

- Gerir as chaves associadas à cifragem ou facilitar a gestão;

Da mesma forma que se organizam dados, é também necessário organizar as chaves que os protegem. A cifragem apenas torna segredos grandes (dados) em segredos pequenos (chaves). Como são pequenos, são fáceis de perder. Devido à natureza do armazenamento de dados, estes pequenos segredos terão um período de vida possivelmente longo, outro fator que os torna ainda mais suscetíveis a serem perdidos. Se este pior caso se verificar, a perda de informação é garantida, e se a informação for perdida, não há forma de a recuperar, pois encontra-se cifrada.

Outro pormenor que torna a gestão de chaves uma necessidade é a possível existência de múltiplos utilizadores no mesmo espaço de armazenamento. Neste caso, é indesejável deixar o processo de gestão a cargo de cada utilizador, pois tal pode constituir um risco para as chaves em si. Por exemplo, um utilizador pode ganhar acesso à chave de outro inadvertidamente, graças à displicência daquele que teve as suas chaves comprometidas. Se existir uma gestão uniforme, torna-se razoável garantir que tal cenário seja pouco provável de acontecer.

Um último problema que merece ser salientado é o facto de um certo conjunto de dados poder existir tanto em estado cifrado com decifrado no mesmo espaço de armazenamento ou em diferentes locais. Isto pode possibilitar ataques às chaves e, consequentemente, a obtenção das chaves em si. Um sistema, para ser considerado seguro, deve tentar garantir que não existem cópias decifradas de dados cifrados.

2.4.3 Granularidade da Cifragem

Dependendo da granularidade da cifragem, este processo realiza-se sobre diferentes objetos. No caso de granularidade de ficheiros, cada ficheiro é cifrado de forma separada, enquanto que no caso de granularidade de disco, os blocos do dispositivo de armazenamento são cifrados individualmente. Em ambos os casos, é assumido o uso de cifragem simétrica.

Quando a granularidade é de ficheiros, estes podem ser cifrados individualmente, cada um com uma chave simétrica única. Se informação for perdida durante o processo de cifragem, então só é perdida informação relativa ao ficheiro em questão, não afetando mais ficheiro algum. Neste nível de granularidade, a gestão de chaves toma uma dimensão elevada. Para aliviar isto, pode-se utilizar a mesma chave para todos os ficheiros, mas se essa chave for comprometida, então todos os ficheiros serão comprometidos, ou seja, não se estará a usar esta granularidade ao seu potencial máximo.

Usar uma só chave para todos os ficheiros cria um grave problema derivado do funcionamento dos modos de operação de cifras. Nestes modos, é necessário utilizar um elemento de começo (IV) para poder realizar a cifragem de um ficheiro inteiro de forma segura. Este elemento tem de ser único para cada ficheiro, pois se não o for, pode acontecer que ao cifrar os mesmos dados em ficheiros diferentes, o mesmo resultado é obtido em todas as operações de cifra. Isto cria padrões de cifragem que um agente mal-intencionado pode usar para prever o conteúdo do resultado da cifra sem recorrer à chave.

A solução para o problema anterior consiste em garantir que, para cada ficheiro, é usado um IV diferente. Porém, tal pode não ser possível de se garantir. Se se utilizar dados aleatórios, o problema do dia de aniversário [40] pode surgir inesperadamente, e se utilizar-se metadados do sistema de ficheiros para a sua criação, não há forma de garantir que esses metadados são únicos para cada ficheiro, fazendo com que o mesmo problema do dia de aniversário surja novamente. Para resolver esta questão, pode-se tentar garantir que cada IV é único (p. ex. utilizar uma sequência) ou usar um só IV para múltiplas chaves garantidamente únicas (recursivo, pois pode não ser possível garantir que as chaves são únicas). De notar que o valor de IV nem sempre necessita de ser secreto, dependendo do modo de operação escolhido.

Um problema deste grau de granularidade é que metadados específicos aos ficheiros não são cifrados, pois não pertencem aos ficheiros em si, mas sim às estruturas de dados do sistema de ficheiros. Este fator é relevante nos serviços *cloud*, pois estes possuem metadados associados a cada objeto. Se estes metadados não forem tratados pelo processo de cifragem, pode-se estar a dar informação a um atacante sobre esses objetos.

No caso em que a granularidade é ao nível de blocos, cada um destes blocos pode ser cifrado individualmente. Isto significa que é possível usar uma chave por cada bloco. Porém, na prática só é usada uma única chave simétrica para todos os blocos de um dado dispositivo. A razão para isso deve-se ao problema de gestão de chaves.

No nível de granularidade de ficheiros, se se utilizar uma chave para cada ficheiro, o tamanho do "chaveiro" seria significativo, mas não o suficiente para ter um impacto relevante no tamanho por ele ocupado. Porém, neste caso, o tamanho do "chaveiro" seria potencialmente na casa das milhões de chaves, o que tornaria a sua gestão muito complexa. O uso de chaves únicas para cada bloco seria necessário se não fosse possível gerar um IV único para cada bloco, o que não é o caso.

Para gerar IVs únicos para cada bloco, pode-se recorrer à utilização do número de bloco. Este garantidamente será único, pois dois blocos não podem ter o mesmo número. Isto significa que não só não é desejável usar chaves únicas para cada bloco, como também não é necessário, resolvendo desta forma o problema e justificando o uso de uma só chave para um dispositivo de blocos inteiro.

Uma vantagem deste nível de granularidade é que não são somente os ficheiros que são cifrados, mas sim tudo o que está no disco, incluindo metadados de sistemas de ficheiros. Este nível é cego quanto a diferenciação de tipos de dados, tratando tudo de forma igual e ao mesmo tempo. Isto, contudo, significa que na eventualidade de se perderem dados, essa perda pode afetar várias coisas em simultâneo, tornando-a mais penosa. No âmbito de serviços *cloud*, ao se utilizar esta granularidade na cifragem, pode-se ignorar os campos de metadados dos *blobs* e usar exclusivamente os do sistema de ficheiros.

Devido a diversas restrições, cada grau de granularidade pode levar ao uso de um modo de operação específico. A escolha deste modo de operação pode ter consequências diferentes para cada granularidade. Um fator de destaque é a possibilidade de acesso aleatório, isto é, ser capaz de aceder partes específicas de um conjunto de dados cifrado. Certos modos de operação obrigam a que todos os dados de um conjunto estejam interligados, o que obriga a ter o conjunto inteiro disponível para aceder somente uma parte deste. Isto é realizado para que os dados originais sejam irrecuperáveis quando qualquer parte dos dados cifrados são indevidamente alterados. Numa granularidade de blocos, é essencial ter acesso aleatório para aceder cada bloco individualmente. Ao trabalhar com ficheiros, pode ser desejável não ter acesso aleatório.

2.4.4 Ferramentas

Dependendo do nível de granularidade que se pretende (ficheiros ou blocos), existem diferentes ferramentas para obter segurança de dados armazenados.

No que toca a granularidade de ficheiros, existem o EncFS [41] e o eCryptfs [42] para o sistema-operativo GNU/Linux. Enquanto o primeiro é implementado em FUSE, o segundo é um módulo de kernel. Quanto a granularidade de blocos, existe o dm-crypt para GNU/Linux, que deve ser usado juntamente com o Linux Unified Key Setup (LUKS) [43], e o Truecrypt [44]. O dm-crypt em si é o componente responsável pela cifragem do dispositivo de blocos, enquanto o LUKS define e aplica um formato aberto para o armazenamento da chave simétrica do processo de cifragem. O Truecrypt é possivelmente o programa com mais exposição, visto ser compatível com múltiplos sistemas-operativos incluindo Microsoft Windows. Porém, o seu desenvolvimento foi subitamente terminado sob circunstâncias dúbias [45]. [46] oferece uma comparação detalhada entre estas soluções.

2.5 Redundância e Distribuição

A questão da disponibilidade nos serviços *cloud* tem duas vertentes: disponibilidade de rede e disponibilidade de dados. Um dos objetivos do sistema que se pretende descrever envolve lidar com ambas estas componentes. A primeira pode ser trivialmente resolvida; o segundo requer conhecimentos de técnicas de redundância atualmente disponíveis. Nesta secção, irão ambos ser abordados.

A manutenção de disponibilidade de uma infraestrutura como a de CSPs não é trivial, como [7] demonstra. Porém, a descrição e fundamentação do uso de técnicas que lidem com problemas do lado dos CSPs não faz parte do âmbito deste documento. Para além disto, também não é relevante discutir o que pode ser feito para mitigar falhas nos recursos locais. Na verdade, o que se pretende abordar é de que forma um programa de um utilizador pode lidar com falhas dos serviços *cloud*.

2.5.1 Falhas

Uma falha consiste na perda de acesso a um recurso remoto por parte de um recurso local. Estas podem acontecer de forma imprevisível se um dos participantes ficar indisponível sem aviso prévio, ou de forma previsível, se a quebra for pré-programada. Em ambos os casos, um recurso que previamente podia ser usado deixa de estar disponível. Estes dois casos podem ser lidados de forma conjunta ou de forma separada, dependendo das técnicas utilizadas.

Tratar de uma falha previsível é um processo facilmente reduzido a uma de duas opções: providenciar recursos para fazer frente à falha ou tratá-la como se fosse uma falha imprevisível. Para lidar com uma falha imprevisível, esta tem de ser detetável. Tanto no caso de problemas de rede como no caso de problemas de dados, existem ferramentas que podem ser disponibilizadas a um utilizador, de forma a permiti-lo reagir.

Comunicações de rede estão dependentes do uso de diversos protocolos. Estes possuem mecanismos de *timeout* que, passado um certo período de tempo sem receber dados que sirvam para a manutenção da ligação, determinam essa mesma ligação como falhada. Estes mesmos protocolos podem conter mecanismos que tentem automaticamente reestabelecer a ligação, sem ser necessária a intervenção do utilizador. Como alternativa, o utilizador pode, de forma programática, detetar essa mesma falha e lidá-la como preferir. Um exemplo de lidar com estas falhas é acumular acessos por realizar numa fila ou estrutura de dados semelhante, para depois realizá-los *a posteriori*.

No que toca a falhas de armazenamento, o processo já pode ser mais complexo. Uma falha deste tipo pode significar diferentes coisas: a infraestrutura onde os dados se encontram está temporariamente indisponível; ocorrência de perda ou modificação indevida de dados. O primeiro caso pode ser detetado de forma semelhante à de falha de comunicação de rede ou através de mensagens de erro que as interfaces dos CSPs estão permitidas a transmitir; no segundo caso, o utilizador pode recorrer a metadados verificados localmente ou, se o CSP for capaz, através de mensagens de erro.

Esta diferenciação entre as falhas de armazenamento é crucial no caso das *clouds*: em sistemas locais, não existe um caso de indisponibilidade de dispositivos de blocos, ou é possível acedê-lo ou não. Isso significa que vai ser sempre necessário proceder a operações de recuperação e reescrita total de dados mal se disponibilize um novo dispositivo de substituição. No caso das *clouds*, esse estado significa que se a situação tratar-se apenas de indisponibilidade temporária, então não vai ser preciso realizar operações de reescrita totais, pois os dados não se perderam por completo, simplesmente não é possível obtê-los por um espaço de tempo indeterminado.

2.5.2 Integridade de Dados

A questão de garantia de integridade de dados pode ser resolvida através do uso de valores MAC. Estes valores, após calculados, podem ser armazenados juntamente com os dados. Quando estes dados forem novamente acedidos, pode-se recalcular o valor de MAC e comparar o novo valor com o previamente calculado. Se forem diferentes, então os dados não se encontram num estado íntegro.

CSPs são capazes de possuir simples valores de *hash* como metadados. Efetivamente, no caso do serviço AWS [14], cada objeto tem associado a si um valor MD5 como metadado, que nada mais é do que um valor de *hash*. No momento de acesso ao objeto, se o valor *hash* não for igual ao valor que pode ser derivado a partir do estado atual do objeto, então o próprio serviço *cloud* pode retornar uma mensagem de erro.

Depender deste tipo de metadados dos serviços *cloud* não é uma opção segura. Um valor de *hash* simples não é o mesmo que um valor MAC. A autenticidade e validade do valor não pode ser confirmada, visto ter sido produzido sem o uso de uma chave simétrica. Outro fator a ter em conta é o desconhecimento de onde ao certo este valor é calculado no interior de um CSP. Dados podem ser movidos no interior da infraestrutura de um CSP, e nesse processo podem ocorrer perdas de dados, o que pode tornar o uso destes valores irrelevante se tiverem sido calculados após uma tal perda.

Para fazer frente a este desafio, um utilizador pode calcular os seus próprios valores de MAC. No entanto, terá de decidir onde irá armazenar estes valores. Essa questão não é difícil de resolver: é indiferente se estes valores são armazenados no recurso local do utilizador ou se são embebidos diretamente no objeto/colocados como campo próprio de metadados. Vejamos porquê.

Se o utilizador decidir armazenar os valores MAC localmente, tem a garantia que esses valores não serão danificados na infraestrutura do CSP. Contudo, isso não torna esses valores isentos de corrupção por parte dos recursos locais. Se um valor de MAC tiver sido corrompido localmente, torna-se impossível verificar a integridade do objeto associado a este, mesmo que o objeto não tenha sido corrompido.

No caso do utilizador guardar esse valor junto com o objeto ou como metadado dele no serviço *cloud*, não temos garantia que esse valor não será corrompido pelos recursos da sua infraestrutura. Ou seja, também se pode, neste cenário, verificar o caso em que o objeto em si não foi corrompido, mas o MAC foi.

Sendo assim, pode-se concluir que ambas as opções apresentam os mesmos problemas, o que torna a escolha de um ou outro indiferente, pelo menos no que toca a basear a escolha apenas nos casos de corrupção, pois esta escolha pode ser vista também por uma perspetiva de disponibilidade.

Armazenar os valores MAC localmente tem a vantagem de permitir ao utilizador verificar a integridade de um objeto que esteja em recursos locais sem ter que recorrer ao CSP. Contudo, se os valores forem guardados nos CSPs, estes ficam disponíveis em qualquer ponto de acesso, tornando-os portáteis. Esta é uma escolha que depende do uso que um utilizador pretende realizar do serviço *cloud*.

2.5.3 Recuperação de Dados

Perante a possibilidade de detetar falhas de integridade, torna-se necessário saber como proceder à recuperação dos dados originais. Para tal, é preciso ter em conta as seguintes condições:

- O nó onde os dados são armazenados é incapaz de realizar qualquer tipo de processamento;
- A solução não deverá ocupar demasiado espaço, mas deve ser capaz de produzir informação extra.

A primeira condição prende-se com o próprio funcionamento dos serviços de armazenamento das *clouds*: um contentor não fornece um ambiente de execução, somente espaço para lá colocar objetos. A segunda condição é derivada da cobrança realizada ao utilizador por espaço gasto no serviço. Quanto menos espaço, mais barato é. Um sistema que pretende introduzir redundância vai ter que ocupar espaço, mas é desejável que não ocupe demasiado.

A forma mais trivial de recuperar dados consiste em simplesmente ter várias cópias dos dados mantidas em simultâneo, distribuídas em vários espaços de armazenamento diferentes. Porém, o custo desta solução é linear: se para cada objeto se quiser ter uma redundância de 2 cópias, então o espaço total gasto para redundância será $2n$, onde n corresponde ao número de objetos. Isto, portanto, não é uma forma eficiente de gerir o espaço.

Como alternativa, existem várias técnicas de introdução de redundância sem replicação, às quais foram dadas o nome de técnicas de codificação. Estas técnicas pegam em dados e dividem-nos em m fragmentos. Esses são depois recodificados em n fragmentos novos, sendo

que $n > m$. Chamemos a $r = \frac{m}{n} < 1$ o rácio de codificação. Um rácio r aumenta o custo de armazenamento por $\frac{1}{r}$. O objetivo destas técnicas é conseguir recuperar os dados originais recorrendo a apenas m fragmentos de um total de n [47].

Um exemplo tradicional deste tipo de técnica é o de cálculo de paridade. Assumindo que existe um objeto a que é divido em l fragmentos de nome x_i, onde $1 <= i <= l$, esta técnica consiste em criar um fragmento extra cujo resultado seja igual a $x_{l+1} = x_1 \oplus x_2 \oplus \ldots \oplus x_l$. Quando se pretende recuperar da falha de um fragmento i, basta calcular $x_i = x_1 \oplus \ldots \oplus x_l \oplus x_{l+1}$ (sendo que x_i em si não é incluído no lado direito da equação). Esta técnica só é resistente à falha de um fragmento. Ou seja, se falharem dois fragmentos ao mesmo tempo, torna-se impossível de recuperá-los [31].

Esta limitação pode ser considerada razoável no contexto de utilização de espaço de armazenamento de *clouds* por parte de um utilizador. Apesar destas possuírem possíveis problemas de disponibilidade, a probabilidade de dois CSPs falharem em simultâneo é baixa. No entanto, se for necessário obter mais garantias de recuperação, pode-se recorrer a técnicas ainda mais poderosas.

Como alternativas ao cálculo de paridade, existe a codificação Reed-Solomon, Cauchy Reed-Solomon e codificações especiais de Redundant Array of Independent Disks (RAID) 6. O funcionamento de cada um destes é complexo, sendo que a explicação de cada um está para além do âmbito deste documento. No entanto, é possível verificar a teoria por detrás de cada um deles em [48] e [49].

Existem dois critérios-chave que devem ser tidos em conta na escolha da técnica a usar: resistência a falha e performance. Cada um destes algoritmos, como mencionado, apresenta um diferente grau de resistência a falha. Para além disso, nem todos consomem o mesmo número de recursos para serem eficazes, tanto em termos de espaço gasto como em termos de número de leituras e escritas. [50] oferece uma comparação entre diferentes técnicas de codificação; [51] oferece um resumo da comparação entre técnicas de paridade e restantes técnicas de codificação.

2.6 Implementações Atuais

Atualmente, [52] é a única implementação que trate o mesmo problema, abordando muitas das mesmas tecnologias descritas neste capítulo. Esta implementação fornece uma *cloud* híbrida, na qual os metadados são guardados localmente. Estes metadados consistem em valores MAC e outras informações necessárias ao seu funcionamento. A gestão de acessos é realizada ao nível do sistema de ficheiros local, sendo a granularidade da cifragem a de ficheiros. Para um ficheiro ser colocado nas *clouds*, utiliza-se uma interface gráfica para efetuar a operação. Antes do ficheiro ser colocado, ele é cifrado e é codificado. Os vários fragmentos do ficheiro são distribuídos pelos CSPs utilizados.

Capítulo 3

Desenho da Arquitetura

No presente capítulo, é discutido o cenário de uso do sistema, o traço geral da sua arquitetura e os seus diversos componentes numa perspetiva de alto-nível.

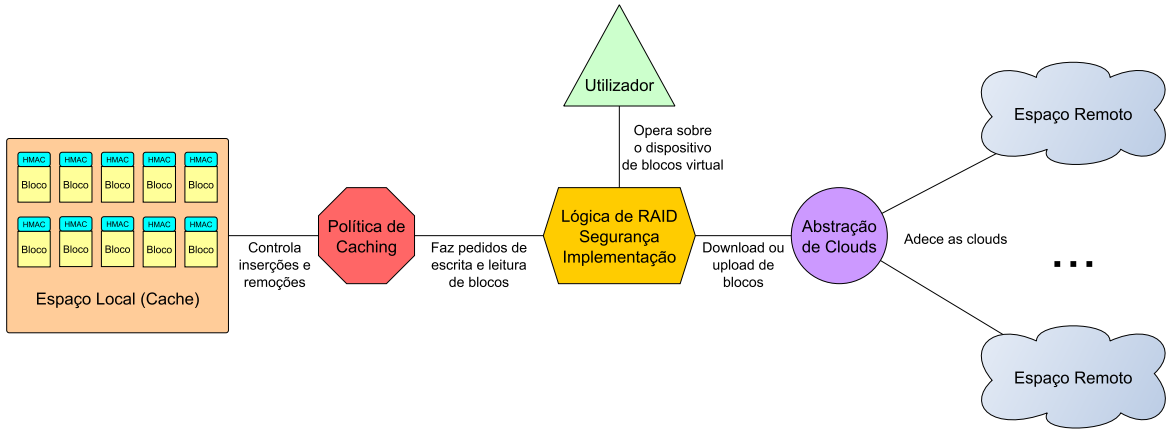


Figura 3.1: Representação Abstrata do Sistema

O utilizador opera sobre o dispositivo de blocos virtual criado pela lógica de RAID e de implementação. Esta lógica tem de lidar com dois componentes intermediários, um para lidar com o espaço local, outro para lidar com o espaço remoto.

3.1 Cenário de Uso

Para o sistema que se desenvolveu e é descrito no presente documento, foi determinado o seguinte cenário de uso:

Uma máquina a desempenhar as funções de servidor, a executar o sistema-operativo GNU/Linux, considerada capaz de manter integridade e segurança local, com acesso direto a recursos computacionais de armazenamento, a fornecer um serviço de cloud *híbrida seguro através de um dispositivo de blocos virtual em* user space *, com as operações de leitura e escrita funcionais, com acesso a três ou mais* clouds *no melhor caso de disponibilidade das* clouds *e com acesso exclusivo ao espaço de armazenamento contratado.*

O dispositivo de blocos virtual realiza as operações de leitura e escrita localmente e no espaço de armazenamento das *clouds*. O espaço local funciona somente como uma cache, gerida por uma política apropriada. O conteúdo é cifrado e a sua integridade é garantida. Comunicações com as *clouds* são abstraídas e operações de escrita são registadas num sistema de *journaling*.

3.2 Componentes Lógicos

O sistema é constituído por quatro componentes lógicos distintos. Na presente secção, procede-se a uma descrição abstrata de cada um deles, descrevendo o funcionamento sem entrar em detalhes práticos de implementação.

Secção 3.2.1 lida com a organização do espaço local, indicando o que é guardado, de que forma e para que serve; secção 3.2.2 trata da forma como os elementos locais são geridos, descrevendo o funcionamento da política de *caching* escolhida; secção 3.2.3 descreve como segurança é obtida; secção 3.2.4 fala de como o sistema obtêm redundância, detalhando a organização dos dados no espaço de armazenamento remoto e a técnica de codificação escolhida.

3.2.1 Armazenamento Local

Inerente à condição de sistema híbrido está a utilização de espaço de armazenamento local. Secção 2.3 indica que a gestão de operações locais pode ser realizada de uma de três formas diferentes: recorrendo ao funcionamento do sistema de ficheiros local; utilizando um sistema de ficheiros próprio; recorrendo a um dispositivo de blocos próprio. Enquanto o funcionamento do

primeiro está adaptado às necessidades do sistema local, as outras duas podem funcionar tendo consciência das limitações e potencialidades do espaço de armazenamento da *cloud*.

Para o sistema implementado, a escolha recaiu sobre um dispositivo de blocos virtual. Esta escolha deve-se a duas principais razões: facilidade de implementação e granularidade fina.

A implementação é simplificada devido ao número de operações a implementar ser mais reduzido e à ausência de estruturas de dados próprias. Há menos operações a implementar, pois um dispositivo de blocos só necessita de ser capaz de ler e escrever conjuntos de bytes, enquanto um sistema de ficheiros tem de implementar várias outras funções (como demonstrado pelo FUSE [53]); sistemas de ficheiros estão dependentes de estruturas de dados que descrevem o seu estado, enquanto um dispositivo de blocos não possui algo equivalente.

A granularidade fina é importante para determinar o que deve ou não ser enviado para as *clouds*, assim como para facilitar a manutenção de redundância. Esta permite evitar enviar *streams* de zeros, poupando assim largura de banda e espaço, e possibilita manter redundância a um nível baixo, dividindo os dados em blocos de tamanho à escolha da implementação.

Sendo o sistema baseado num dispositivo de blocos, torna-se necessário formatá-lo com um sistema de ficheiros para se poder utilizá-lo. A realização de uma escolha apropriada neste caso deve ter em conta o número de acessos que o sistema de ficheiros efetua por cada operação e onde armazena as suas estruturas de dados. Estes fatores são relevantes no contexto do presente trabalho, pois os acessos ao espaço de armazenamento remoto acarretam tempos de execução elevados e variáveis e o tamanho das estruturas, juntamente com a sua localidade, podem ter penalizações relevantes. Esta escolha é deixada ao critério do utilizador. Ao implementar um dispositivo de blocos próprio, este é um ponto que é abstraído da descrição do sistema.

Estabelecido que a granularidade do sistema é de blocos, torna-se necessário definir o que é um bloco. Esta unidade irá conter um conjunto de bytes. O tamanho de um bloco é variável, mas tem um tamanho máximo ao critério do utilizador. O bloco só existe se tiver um tamanho maior que zero, impondo, desta forma, um tamanho mínimo também. O tamanho variável irá permitir ao utilizador poupar espaço nas *clouds*, enquanto o limite máximo é necessário para a manutenção de redundância de dados. O tamanho de um bloco não está associado ao tamanho de bloco do sistema de ficheiros escolhido pelo utilizador. A figura 3.2 apresenta a estrutura de um bloco.

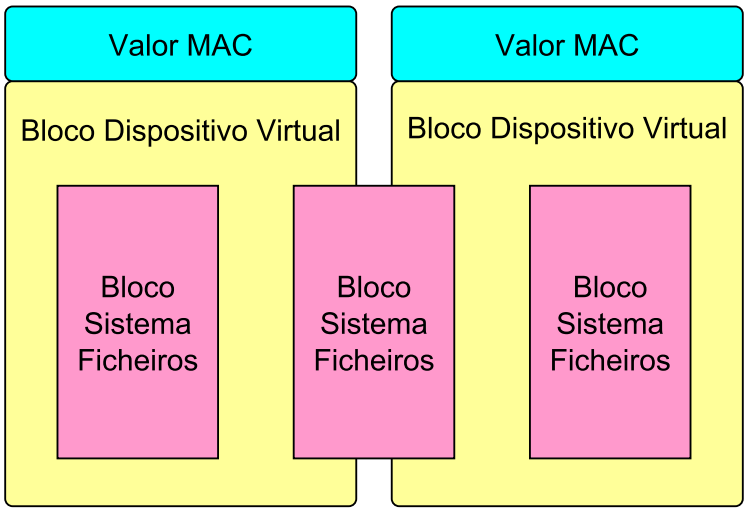


Figura 3.2: Estrutura de Blocos do Dispositivo

Este diagrama demonstra um de vários estados possíveis para os blocos do dispositivo virtual implementado. Neste caso, o tamanho de bloco do sistema de ficheiros é menor que o dos blocos do dispositivo e ambos os blocos do dispositivo estão cheios (atingiram tamanho máximo). No final de cada bloco do dispositivo é colocado o correspondente valor MAC, para verificação de integridade. De notar o bloco do sistema de ficheiro que se encontra armazenado em dois blocos diferentes do dispositivo.

A cada bloco são associados dois elementos de metadados: um valor de MAC e um número de identificação. O primeiro é necessário para garantir a integridade dos dados armazenados na *cloud* de forma criptograficamente segura. O número de identificação é atribuído sequencialmente tal como num dispositivo físico, começando no número 1, permitindo, assim, referenciar um bloco de forma única. Um pormenor da numeração que se tornará mais claro na secção 3.2.4 é a atribuição de um número aos blocos de redundância. A atribuição sequencial destes números é o método utilizado no sistema, visto que consegue garantir um número único para cada bloco de forma trivial e eficaz.

A utilização do espaço local como *cache* implica que nem todos os blocos são armazenados localmente, sendo somente um sub-conjunto deles mantido com o propósito de melhorar os tempos de acesso através da exploração de localidade temporal e espacial. A existência de uma cache significa também que são mantidos metadados relativos ao seu funcionamento. Estes metadados da cache não são armazenados nas *clouds*, pois são usados somente durante a execução do sistema e são relativos exclusivamente ao estado do espaço local. Se este mesmo espaço tiver conteúdo e os metadados não estiverem presentes, o funcionamento do sistema pode ser perturbado.

A existência de uma *cache* significa que o espaço local e o espaço remoto poderão atingir estados diferentes. A ordem de acesso a blocos é, em primeiro lugar, o espaço local, ficando em segundo o espaço remoto. Os metadados da cache irão descrever quais blocos estão em estado diferente daquele que se verifica para o mesmo bloco no espaço remoto. Como tal, se não se tiver os metadados de cache, perde-se essa informação, o que pode levar à perda de dados através da atuação da política de cache (esta pode eliminar um bloco que foi modificado somente localmente).

Existem dois tipos diferentes de cache: *write-through* e *write-back* [54]. Na primeira, todas as operações feitas na cache são realizadas no destino (neste caso as *clouds*) assim que possível; na segunda, as mesmas operações são diferidas até ao momento em que um dado elemento (neste caso um bloco) tem de ser eliminado. A cache deste sistema funciona como *write-through*, de forma a ter os dados sempre atualizados no espaço remoto (que é o principal espaço de armazenamento do sistema).

O espaço local armazena ainda registos de operações de escrita pendentes no espaço remoto. Estes registos são criados e mantidos pelo componente de *journaling*. Cada *cloud* utilizada pelo sistema tem um registo diferente, sendo o tamanho destes variável ao longo da execução.

3.2.2 Gestão Local

O funcionamento do espaço local como *cache* tem, como propósito, melhorar tempos de acesso. Para a gestão de blocos neste espaço, é necessário recorrer a uma política de *caching*. Esta determina o grau de importância da localidade temporal e espacial e define ainda como essas localidades devem ser exploradas. A função desta é, portanto, indicar qual e quando um bloco deve ser eliminado.

Uma entrada da *cache* pode estar num dos seguintes três estados: vazia; preenchida e inalterada; preenchida e alterada. O estado de uma entrada preenchida tem associada a si um elemento de metadado que indica se se encontra alterada ou não.

No nosso sistema, as entradas de blocos ocorrem quando o utilizador ou a lógica do sistema necessitam dele para uma operação de leitura ou escrita. Nesse momento, o sistema verifica se o bloco se encontra na cache. Se não o encontrar no espaço local, vai até à *cloud* correta e obtém dela o bloco. O processo de obtenção consiste em descarregar o bloco e colocá-lo neste espaço.

O processo de saída está dependente do funcionamento da política de *caching*. Se esta determinar que a presença de um dado bloco inalterado já não é útil na exploração de localidade, então é eliminado. Tal verificação por parte da política só é realizada quando a cache se encontra completamente cheia, evitando assim operações de saída preventivas.

Existem inúmeras políticas de *caching* [55], cada uma com diferentes esquemas de aproveitamento dos dois diferentes tipos de localidade. Para o nosso sistema, recorreu-se a uma política de Least Recently Used (LRU) que, tal como o nome indica, consiste em remover o elemento usado menos recentemente. Para uso desta política, é necessário recorrer a uma estrutura de dados que permita retirar elementos aleatoriamente do seu interior e que respeite a ordem de entrada dos elementos, ou seja, uma fila. Nesta estrutura, a política coloca, por ordem temporal de uso, os blocos usados mais recentemente no seu fundo e os usados menos recentemente à cabeça.

O sistema recorre a duas filas em simultâneo para a política LRU, uma para blocos que não foram modificados, de nome fila m, e outra para espaços vazios na cache, com o nome fila c. A fila m contém blocos candidatos a serem eliminados sem que haja perda de informação, a fila c permite evitar a pesquisa por entradas vazias no momento em que se pretende colocar um novo bloco na *cache* e facilita a implementação da política. Como alternativa, seria possível utilizar somente uma fila, desde que os elementos das duas filas fossem representados internamente de forma igual.

Quando um bloco entra na cache, ele é colocado no fundo da fila m. Se o bloco for entretanto modificado, mas não for imediatamente copiado para a *cloud*, ele é removido da fila m, de forma a evitar perda de dados. Quando o bloco for entretanto copiado para a *cloud*, ele é novamente inserido na fila m. Se for necessário um espaço na cache para um bloco novo e a cache estiver cheia, remove-se um dos blocos da fila m e usa-se o espaço que lhe pertencia para lá colocar o novo bloco.

Quando a cache não se encontra cheia, recorre-se à fila c. O ingresso de entradas novas nesta fila ocorre no momento de arranque do sistema (se a cache ainda tiver espaços vazios) e, no caso da implementação recorrer a algum tipo de bloco especial não especificado neste capítulo, após a eliminação de tais blocos. O segundo caso verifica-se na implementação descrita no capítulo seguinte, sendo pormenorizado na secção 4.5. A eliminação de entradas desta fila c ocorre quando é necessário um espaço vazio e a cache não está cheia.

O uso da política LRU é considerado razoável, na medida em que é capaz de se adaptar a várias situações com resultados aceitáveis [31]. Existem várias políticas alternativas, cada uma com graus de eficiência diferentes dependendo do sistema onde se inserem, mas a escolha desta foi determinada como suficiente para o que se pretendia atingir. Secção 5.7 apresenta uma perspetiva mais prática relativamente à escolha desta política.

3.2.3 Segurança

O componente lógico encarregue da segurança do sistema lida com duas questões separadamente, nomeadamente a questão de confidencialidade e a de manutenção de integridade. A primeira é abordada através do uso de técnicas de cifragem tendo em conta o cenário próprio de dispositivos de blocos, a segunda pelo uso de valores MAC produzidos separadamente do processo de cifragem.

3.2.3.1 Cifras e Modos de Segurança

Para obter confidencialidade, recorre-se a uma cifra em conjunto com um modo de operação. A escolha de ambos destes elementos foi parcialmente baseada em elementos objetivos, nomeadamente se da sua utilização resultam ou não perdas de dados e se da criptanálise destas não surgiram falhas significativas que coloquem em risco a confidencialidade dos dados. Elementos subjetivos que influenciaram esta escolha consistem na qualidade das criptanálises realizadas e do número de casos de uso destas, critérios estes que, no campo da criptografia, devem ser tomados em conta (quanto mais uso, mais criptanálises e de maior qualidade terá). Infelizmente, a discussão técnica das opções disponíveis é um processo demasiado elaborado e envolvido para o presente documento. Como tal, a discussão é restringida à justificação da escolha efetuada.

No caso do nosso sistema, a cifra tem necessariamente de ser de blocos. Atualmente, o principal padrão da indústria é a cifra Advanced Encryption Standard (AES) (Rijndael). A definição desta como padrão foi realizada pelo National Institute of Standards and Technology (NIST) após um processo longo de escolha. A definição da cifra encontra-se em [56] e o processo de escolha entre uma seleção de concorrentes está documentado em [57]. Tendo em conta a sua segurança, a sua descrição aberta e a sua utilização intensa, esta é a cifra que o sistema recomenda a utilizar.

Uma implementação do sistema não deve tratar desta cifra de bloco. Como o sistema consiste num dispositivo de blocos virtual, o utilizador não tem garantias logo à partida de confidencialidade. Para conseguir este fator, tem de recorrer a uma ferramenta de cifragem deste

tipo de dispositivos como as mencionadas previamente na secção 2.4.4. A escolha da cifra está a cargo do utilizador, estando o sistema capaz de somente recomendar uma.

A implementação de elementos criptográficos é complexa e empresta-se facilmente a erros de programação. [58] indica que um dos alvos principais de ataques são as implementações destes elementos. Como tal, o uso de ferramentas que são usadas por múltiplas entidades, com uma maturidade considerável e com uma criptanálise extensa, é considerado mais seguro que a criação de uma nova implementação. Isto aplica-se tanto a ferramentas externas como a bibliotecas de programação e não só aos algoritmos de cifragem como também aos modos de operação.

A escolha de um modo de operação é necessária, pois o tamanho de um bloco do nosso sistema é maior que uma palavra da cifra AES. Como foi previamente mencionado, o produto do funcionamento deste não pode ser maior que o espaço ocupado pelos dados originais. Esta restrição não está relacionada com a utilização das *clouds* (apesar de trazer benefícios neste caso), tendo tido origem no funcionamento do *hardware*: num dispositivo físico, um bloco tem um tamanho máximo inflexível. Duas outras restrições, sendo estas específicas aos modos de operação, é a forma como é inicializado, ou dito de outra forma, como o valor do IV é obtido e ainda ter que permitir acesso aleatório aos dados.

Seguindo estas restrições adicionais para os modos de operação, o sistema recomenda o uso do modo XEX-based Tweaked-codebook mode with ciphertext Stealing (XTS) ou Cipher Block Chaining (CBC) com IVs gerados por Encrypted Salt-Sector Initialization Vector (ESSIV). No que toca a XTS, a inicialização de IVs é trivial, pois usa o número de bloco para tal, e a forma como opera garante que qualquer alteração aos dados cifrados resultam em dano total dos dados originais, impedindo tentativas de recuperação de dados sem recorrer à chave. Para além disso, o resultado da sua utilização tem o mesmo tamanho que os dados originais. Está definido como um padrão em [59] e a sua descrição encontra-se em [60]. O CBC é vulnerável a ataques se o IV não for gerado de forma imprevisível. Como tal, recorre-se a técnicas para o tornar seguro, técnicas como o ESSIV. O funcionamento do modo CBC encontra-se descrito em [39] e o processo ESSIV, assim como os mencionados ataques, são elaborados em [61]. Ambos os modos de operação são equivalentes no que toca a garantias de segurança oferecidas, o que leva à recomendação da utilização de uma destas duas.

Nas ferramentas para dispositivos de blocos mencionadas na secção 2.4.4, dm-crypt disponibiliza várias cifras diferentes, mas somente dois modos de operação, os mencionados na presente secção. O Truecrypt disponibiliza também várias cifras, mas somente o modo de operação XTS. No cenário de uso do nosso sistema, a escolha recaiu sobre a ferramenta dm-crypt.

3.2.3.2 Integridade

As garantias de integridade são fornecidas através do uso de valores MAC, resumos seguros de tamanho fixo de conjuntos de dados de tamanho aleatório. Algoritmos de geração destes valores devem garantir que o valor que produzem é único para um conjunto de dados e que a relação entre dados originais e valor produzido é indecifrável. Se existirem ataques que coloquem estas duas restrições em risco, um algoritmo é considerado impróprio para uso no sistema. Os mesmos critérios subjetivos usados nas cifras são tidos aqui em conta.

Tal como no caso do algoritmo de cifra, o sistema não lida com a implementação deste, não se procedendo, então, a uma descrição detalhada do algoritmo escolhido. Porém, o sistema não delega a utilização deste a uma ferramenta externa, integrando uma implementação pertencente a uma biblioteca segura (analisada e validada criptograficamente). A verificação de integridade está dependente da definição do nosso sistema do que um bloco de dados é, o que obriga a que o próprio sistema lide diretamente com a tarefa.

Os algoritmos em questão geram valores de tamanho fixo, na casa dos 256 ou 512 bits de tamanho. Como mencionado, cada bloco tem um valor MAC próprio. Isto significa que o custo que acarretam em termos espaciais é desprezável.

A separação entre confidencialidade e integridade neste sistema deriva do facto de a primeira recorrer a modos de operação que não geram automaticamente os valores MAC. Um exemplo de um modo que une ambas estas questões e permite acessos aleatórios é o Galois Counter Mode (GCM) [62]. Porém, nenhuma das ferramentas consideradas para cifragem disponibiliza este modo porque, no caso de dispositivos de blocos, o produto da cifragem não pode ser maior que os dados originais, algo que seria difícil de garantir quando se produz dados adicionais (valor MAC) para cada bloco.

O algoritmo utilizado pelo sistema é o Secure Hash Algorithm (SHA)2 512/256, com *output* de 512 bits que é truncado para 256. Este algoritmo foi definido como um padrão, novamente pelo NIST, e é usado como tal. Como foi mencionado no caso de cifras, a existência de um

padrão implica que a validade do algoritmo foi já verificada múltiplas vezes por várias entidades diferentes, e esse é um fator importante no processo de escolha. A definição deste algoritmo encontra-se em [63].

A segurança deste algoritmo já foi posta em causa em múltiplas ocasiões [64][65], tendo já sido mesmo definido um novo padrão pela mesma entidade. O SHA3 [66] tem como propósito a substituição direta do SHA2. No entanto, devido à falta de implementações de confiança, foi determinado que o uso de SHA2 seria suficiente. Apesar do algoritmo estar enfraquecido, ainda não foi atacado em condições de pleno funcionamento.

O funcionamento e utilização de valores MAC no sistema procede-se da seguinte forma:

- Cada bloco terá um valor MAC associado a si;
- O valor MAC é calculado para um dado bloco no momento em que o conteúdo do bloco é modificado;
- Este valor é guardado no final de um bloco;
- Um bloco só tem um valor MAC se tiver um tamanho maior que zero;
- Este valor nunca deverá ser eliminado do bloco exceto quando o bloco ficar reduzido a tamanho zero.

3.2.4 Redundância e Armazenamento Remoto

Para oferecer garantias de constante disponibilidade dos dados à guarda do serviço, é necessário recorrer a técnicas de obtenção de redundância. Estas técnicas, no âmbito deste sistema, permitem lidar com potenciais falhas de disponibilidade das *clouds*, sejam elas de disponibilidade ou de perda de dados. Certas destas técnicas requerem que os dados sejam organizados de forma específica para funcionarem corretamente, um pormenor que teve de ser tido em conta no desenvolvimento do nosso sistema.

Para tratar o problema de redundância, o sistema recorre a técnicas de codificação. Estas são preferíveis a redundância pura (isto é, cópias completas de informação), pois permitem um gasto mais reduzido de espaço. De todas as alternativas expostas na secção previamente mencionada, a opção a que o sistema recorre é a de cálculo de paridade. A sua implementação simples e a baixa probabilidade de mais de um CSP ficar indisponível foram os principais fatores nesta escolha.

Esta técnica tem certas implicações de desempenho. No seu normal funcionamento, o valor de paridade é recalculado sempre que um dos conjuntos de dados (bloco) é modificado. De resto, é relevante mencionar que cada operação de escrita implica não só a atualização do bloco modificado, como também do bloco de paridade.

Esta técnica de paridade é geralmente associada ao funcionamento dos esquemas Redundant Array of Independent Disks (RAID) 4 e 5. Um esquema RAID consiste no uso de múltiplos discos rígidos para a obtenção de redundância. A forma como os dados são distribuídos depende do número de discos envolvidos e do tipo de RAID que se pretende. Diferentes esquemas RAID oferecem diferentes tipos de redundância [31]:

- **RAID 0** Dois ou mais discos são dispostos de forma sequencial, fornecendo distribuição de dados em vez de redundância;
- **RAID 1** Dois ou mais discos iguais são emparelhados de tal forma a que o conteúdo do disco principal esteja completamente reproduzido nos discos secundários. Oferece redundância simples;
- **RAID 4** Três ou mais discos são aglomerados e divididos em linhas. Uma linha consiste num conjunto de blocos com número de elementos igual ao número de discos, na qual os blocos envolvidos possuem o mesmo número de bloco no disco a que pertencem. Por exemplo, a primeira linha terá todos os bloco zero de todos os discos. A redundância é obtida através do cálculo de paridade, sendo este cálculo efetuado para todas as linhas. Um disco é dedicado ao armazenamento dos valores de paridade, valores esses que ocupam um bloco inteiro (visto serem o resultado da realização de operações XOR envolvendo os restantes blocos da linha);
- **RAID 5** Igual ao anterior, mas os blocos de paridade são distribuídos de forma uniforme pelos diferentes discos;
- **RAID 6** Funcionalmente igual aos anteriores, mas com um requisito mínimo de quatro discos. Em vez de um valor de redundância por linha, tem dois, tornando o esquema mais resistente a falhas.

O sistema baseia-se no funcionamento de um RAID do tipo 5. Neste, cada disco é equiparado a uma *cloud*. Ou seja, em vez de ser um agrupamento de discos, é, na verdade, um agrupamento de *clouds*. Colocando de parte desde já os RAIDs do tipo 0 e 1 por serem inadequados para

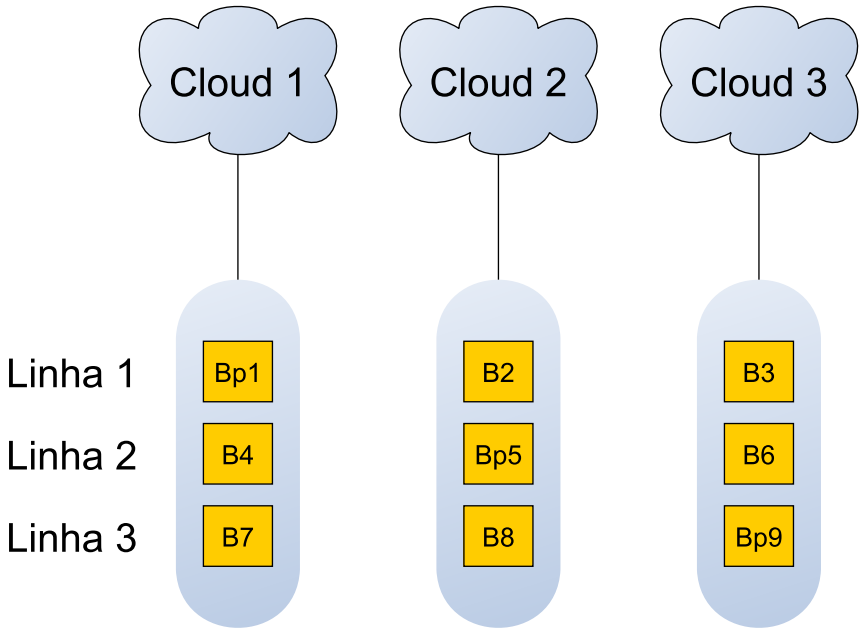


Figura 3.3: Diagrama RAID 5 do Sistema

Estrutura do sistema de RAID. Blocos com nome B_p são blocos de paridade, sendo os restantes blocos de dados. O número de identificação dos blocos é partilhado entre blocos de paridade e de dados. Blocos de paridade são distribuídos na diagonal, da esquerda para a direita. De 3 em 3 linhas, os blocos de paridade voltam a começar a ser inseridos a partir do CSP mais à esquerda.

o que o sistema pretende obter (0 não fornece redundância, 1 é redundância pura), torna-se necessário justificar a escolha de este esquema em específico.

RAID 4, apesar de semelhante ao RAID 5, tem a desvantagem de como todos os valores de paridade estarem no mesmo disco (ou *cloud* para efeito deste sistema), a quantidade de operações de atualização do conteúdo deste disco acaba por se tornar um problema, pois cada modificação em um bloco de uma linha obriga a recalcular o valor de paridade. Este problema seria exacerbado neste sistema, pois a latência de acesso a uma *cloud* tornaria a situação ainda pior. Se estes valores forem distribuídos, torna-se possível paralelizar as escritas destes.

RAID 6 apresenta, como principal vantagem, a capacidade de resistir à falha de dois discos no máximo, em comparação com o limite de uma só falha nos esquemas RAID 4 e RAID 5 (visto estes dois possuírem apenas um elemento de paridade). Porém, isto implica um gasto maior de espaço (mais blocos de paridade), assim como maiores tempos de atualização de paridade.

Se num RAID 6 existem dois elementos de paridade por linha, isso significa que, para cada operação de escrita, é necessário atualizar o bloco(s) de dados modificado(s) e dois blocos de paridade. No RAID 5, só é necessário atualizar um único bloco de paridade. Isto significa que a manutenção de paridade no RAID 6 implica o dobro do número de transações. Isto pode não parecer significativo, mas como foi já mencionado, o utilizador pode ver a sua utilização das *clouds* a ser cobrada por número de transações. Como foi assumido que a falha de mais que uma *cloud* é pouco provável, este incremento no número de operações foi considerado desnecessário.

Os blocos não são modificados quando são transferidos para o espaço remoto. A cada bloco corresponde uma *cloud* em específico, devido à disposição de blocos inerente ao RAID 5. A numeração dos blocos no espaço remoto e no espaço local mantém-se igual, tendo ambas em conta os blocos de paridade. Como mencionado, a numeração é sequencial, tendo início no número 1. No entanto, como a figura 3.3 deixa claro, esse número pertence ao bloco de paridade da linha inicial.

A determinação do número de um bloco de dados terá que ter em conta este pormenor, saltando os blocos de paridade aquando do cálculo. Acessos a um dispositivo de blocos são realizados a partir de valores de *offset* que começam no byte 0 do dispositivo. Com os blocos de paridade presentes no armazenamento, é necessário realizar um *shift* desse valor de *offset*, de forma a que a sua tradução para número de bloco resulte num número associado a um bloco de dados em vez de um bloco de paridade.

Capítulo 4

Desenvolvimento

Esta secção está dedicada à descrição da estrutura da implementação realizada, assim como o funcionamento de cada componente seu. A implementação consiste num servidor NBD, configurável através de um ficheiro de configuração. A interação entre este e os serviços *cloud* é mediada pela camada de abstração Deltacloud. As interações de escrita são registadas em múltiplos *journals*. Redundância é garantida através do uso de um esquema RAID 5, integridade é obtida através do cálculo e uso de valores MAC e confidencialidade é fornecida pela ferramenta dm-crypt.

4.1 Estrutura da Implementação

Os diversos componentes lógicos da implementação não são executados todos de forma sequencial. Na verdade, estes componentes estão divididos entre duas *threads* diferentes implementadas através da biblioteca pthreads, cada uma com responsabilidades e capacidades únicas. Estas são a **thread principal** e a **worker thread**.

À *thread* principal estão associados os componentes NBD, leitura de ficheiro de configuração, ler conteúdo das *clouds*, registo de operações de escrita (*journaling*), manipulação da cache e manutenção de integridade; à *worker thread* foram dadas a capacidade de escrita nas *clouds* de acordo com o conteúdo dos *journals*, manipulação da cache, cálculo de paridade (redundância), manutenção de integridade e do sistema RAID.

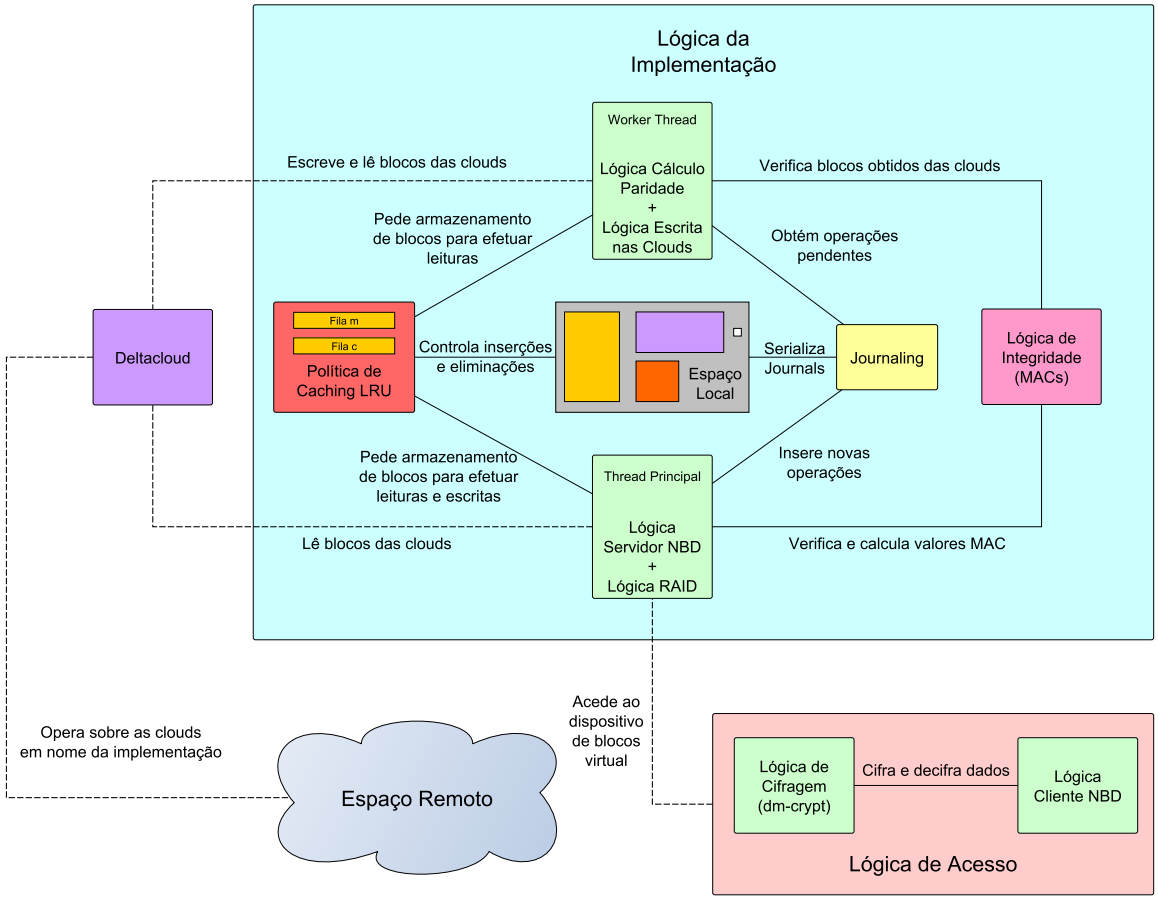


Figura 4.1: Representação de Implementação do Sistema

A tracejado estão comunicações TCP; em linhas sólidas estão comunicações diretas.

A *worker thread*, para realizar operações de escrita nas *clouds*, cria um conjunto de *threads* filhas, chamadas *threads* de *upload*, uma para cada *cloud*, de forma a paralelizar estas mesmas operações. O seu funcionamento está dependente de um contador de tempo. Após o expirar de um período de tempo pré-definido, esta *thread* prossegue a sua execução, execução essa que consiste em verificar se existem operações de escrita nas *clouds* pendentes e, no caso de existirem, recalcular a paridade das linhas envolvidas, escrever nas *clouds* o conteúdo novo, incluindo os blocos de paridade, e eliminar todas as operações da lista que tiveram sucesso.

4.1.1 Ineficiências

Esta implementação utiliza ferramentas externas, isto é, que não fazem parte do código desenvolvido. Duas destas, o Deltacloud e o NBD, requerem o uso de comunicações interprocessuais através de uma ligação TCP. Este tipo de comunicação leva à introdução de latências, das quais

não é possível escapar sem proceder à alteração do funcionamento das ferramentas usadas. Como tal, tiveram que ser assumidas, obrigando a que a implementação fosse o mais eficiente possível no restante do seu funcionamento, de forma a minimizar o impacto que estas latências podem ter. A figura 4.2 ilustra o número de ligações TCP e comunicações interprocessuais necessárias para obter dados armazenados numa *cloud*.

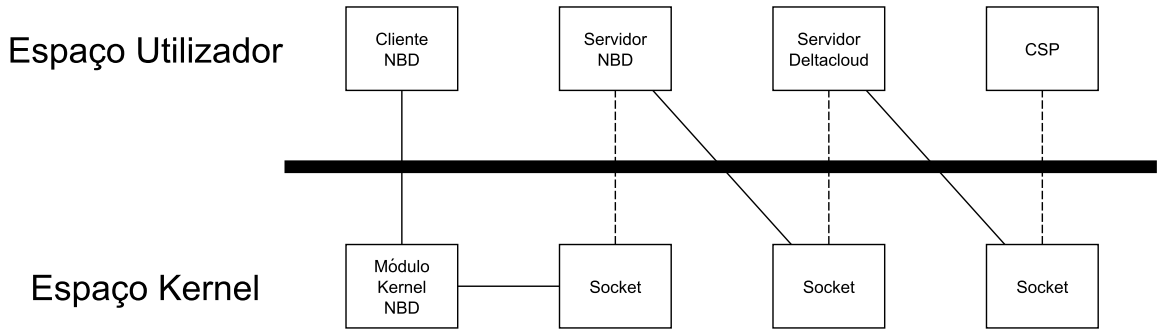


Figura 4.2: Comunicação entre Componentes

Descrito aqui está o processo de envio de um pedido para um CSP. Com uma linha sólida, encontram-se representadas comunicações interprocessuais sem *sockets*, enquanto as linhas a tracejado são comunicações através de *sockets*.

4.2 NBD e Suas Funções

Network Block Device (NBD) é o nome dado à tecnologia incorporada no *kernel* de Linux que permite montar discos remotos localmente [37]. O uso desta tecnologia para a implementação é fulcral, pois é o que permite a criação de um dispositivo de blocos em espaço de utilizador. O processo de montagem de discos remotos é realizado graças a dois componentes que constituem esta mesma tecnologia: o cliente e o servidor. Estes dois componentes não necessitam de residir na mesma máquina. Porém, no cenário de uso desta implementação, ambos os componentes devem residir localmente. O cenário de uso alvo indica que o objetivo é ter a implementação a ser executada numa máquina servidor, mas isso não significa que o dispositivo de blocos virtual da máquina tenha de ser partilhado através desta tecnologia. A figura 4.3 demonstra um possível cenário de uso.

A razão pela qual esta tecnologia permite a criação de dispositivos de blocos virtuais em espaço de utilizador deve-se à forma como a sua implementação foi realizada. O cliente funciona através de um módulo de kernel que acede a um servidor através de um protocolo próprio. Graças

à exclusividade deste protocolo, o módulo kernel pôde ser implementado de forma genérica, de tal forma que, para este, a implementação do servidor é irrelevante, desde que o que é pedido pelo utilizador seja obtido. Visto que o servidor é um programa simples em espaço de utilizador, e que o servidor só tem de servir os pedidos do cliente descritos pelo protocolo próprio, então a descrição programática deste servidor pode ser o que se desejar.

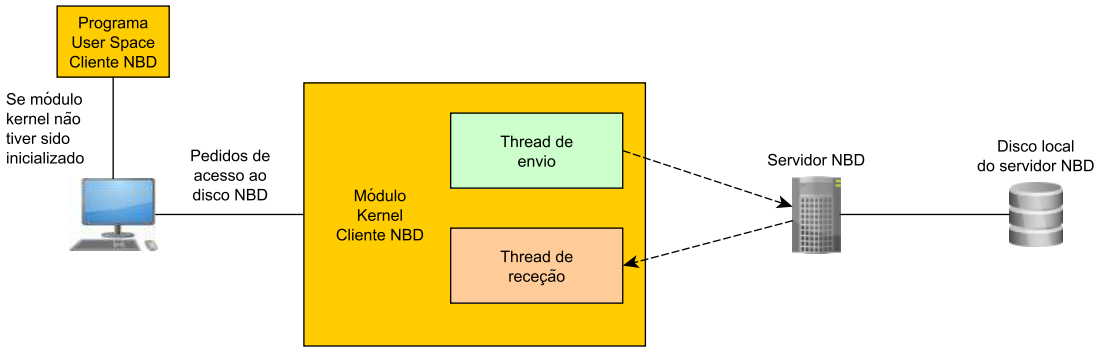


Figura 4.3: Funcionamento do NBD

Funcionamento típico do NBD: No cliente, o programa em espaço de utilizador inicializa o módulo kernel, que funciona com duas *threads*; o servidor recebe os pedidos e responde a eles enviando os bytes pedidos pelo cliente. Linhas sólidas representam comunicações sem *sockets*, tracejadas são com *sockets*.

4.2.1 Cliente

Um cliente NBD é composto por duas peças fundamentais: código em espaço de utilizador e o módulo kernel, o primeiro servindo apenas para inicializar o segundo. Como tal, o funcionamento deste está descrito no módulo kernel e a sua execução realiza-se em espaço de kernel. Para a realização da implementação, não foi necessário fazer quaisquer tipos de alterações ao módulo em si.

Na máquina onde o cliente está a ser executado, o módulo instancia um dispositivo de blocos não-rotacional virtual (ou seja, um disco Solid State Drive (SSD) virtual) com 9 partições, tendo estas o rótulo `/dev/nbdX`, onde $X = [0, \ldots, 9]$, sendo que $X = 0$ corresponde ao dispositivo em si. O módulo lida com cada partição de forma individual, isto é, o funcionamento de cada partição é independente do das restantes. No momento de ativação de uma das partições do dispositivo através duma chamada `ioctl()` realizada pelo código em espaço de utilizador, o módulo cria duas *threads*: uma lida com o envio de pedidos para o servidor NBD, enquanto a outra lida com a receção de respostas desses mesmos pedidos. Isto significa que **cada operação**

realizada sobre uma partição, seja ela de leitura ou escrita, **é lidada à vez**. Este modo de operação, semelhante ao *pipelining* de HTTP 1.1, tem vantagens claras: o cliente pode enviar vários pedidos de uma só vez sem ter que esperar pela resposta de cada um deles e a ordem dos pedidos mantém-se inalterada, evitando assim problemas de concorrência.

4.2.2 Servidor

O servidor NBD é executado em espaço de utilizador, visto que tudo o que este componente faz é responder a pedidos de leitura e escrita de dados recebidos na sua *socket*, semelhante a um servidor *web*. O funcionamento do dispositivo de blocos é descrito pelo servidor, servindo este como *driver* do dispositivo em si.

Um servidor NBD pode implementar várias funcionalidades típicas de dispositivos de blocos, assim como algumas específicas do NBD: leitura, escrita, desconetar, *flush* e Trim Command (TRIM), sendo a última uma funcionalidade tipicamente associada a discos SSD (como foi previamente dito, o disco NBD é apresentado como sendo não-rotacional). Nenhuma delas é obrigatória, mas leitura e escrita são funções cuja implementação é importante.

Cada pedido do cliente é lidado à vez, isto é, um pedido não despoleta a criação de uma nova *thread* só para lidar com ele. A razão para isso prende-se com o funcionamento do módulo de kernel do cliente: como este lida com os pedidos de forma sequencial, seria uma má ideia introduzir concorrência às respostas a dar a estes pedidos (relembra-se que os pedidos não possuem número de sequência).

4.2.3 Considerações Conjuntas

É necessário indicar certos elementos comuns a ambas as funções, nomeadamente a execução da função de obtenção de blocos e como é garantido acesso exclusivo aos blocos que estas funções acedem.

Os blocos são organizados num esquema de RAID 5. Visto que tanto a *thread* principal como a *worker thread* podem manipular blocos ao mesmo tempo, é necessário garantir que uma não interrompe a outra eliminando ou modificando blocos. A solução encontrada foi bloquear a linha a que o bloco pertence sempre que se pretende aceder um bloco. A alternativa seria bloquear cada bloco de forma individual, mas existem alguns argumentos que levaram à escolha de linhas,

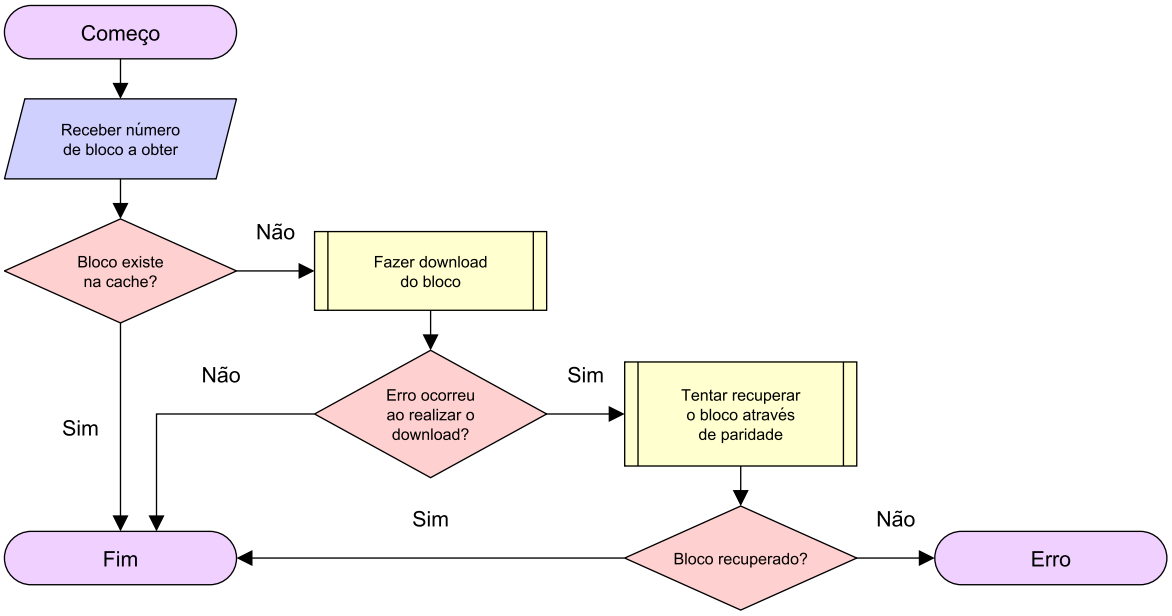


Figura 4.4: Obtenção de bloco

sendo o principal o facto de uma linha ser a unidade-base para o cálculo da paridade. A secção 5.9 apresenta uma discussão mais detalhada do assunto.

Quanto ao processo de obtenção de um bloco por parte destas funções, este é praticamente igual para ambas, podendo-se facilmente abstrair a sua descrição da forma como se encontra na figura 4.4. Ambas as operações de leitura e escrita são bloqueantes para a *thread* principal.

4.2.4 Leitura

A operação de leitura trata de ler um dado número de bytes a partir de um dado *offset*. O resultado desta operação é um *buffer* alocado dinamicamente que, de seguida, é transmitido através de TCP para o módulo kernel do cliente de NBD. A função *Obtenção de bloco* encontra-se descrita na figura 4.4.

No que toca a concorrência, as seguintes condições são garantidas no momento de execução da operação de leitura:

- Não deverá ser possível escrever sobre um bloco ao mesmo tempo que este estará a ser lido;
- Deverá ser possível garantir que o que está a ser lido é, na verdade, a cópia mais recente do bloco;

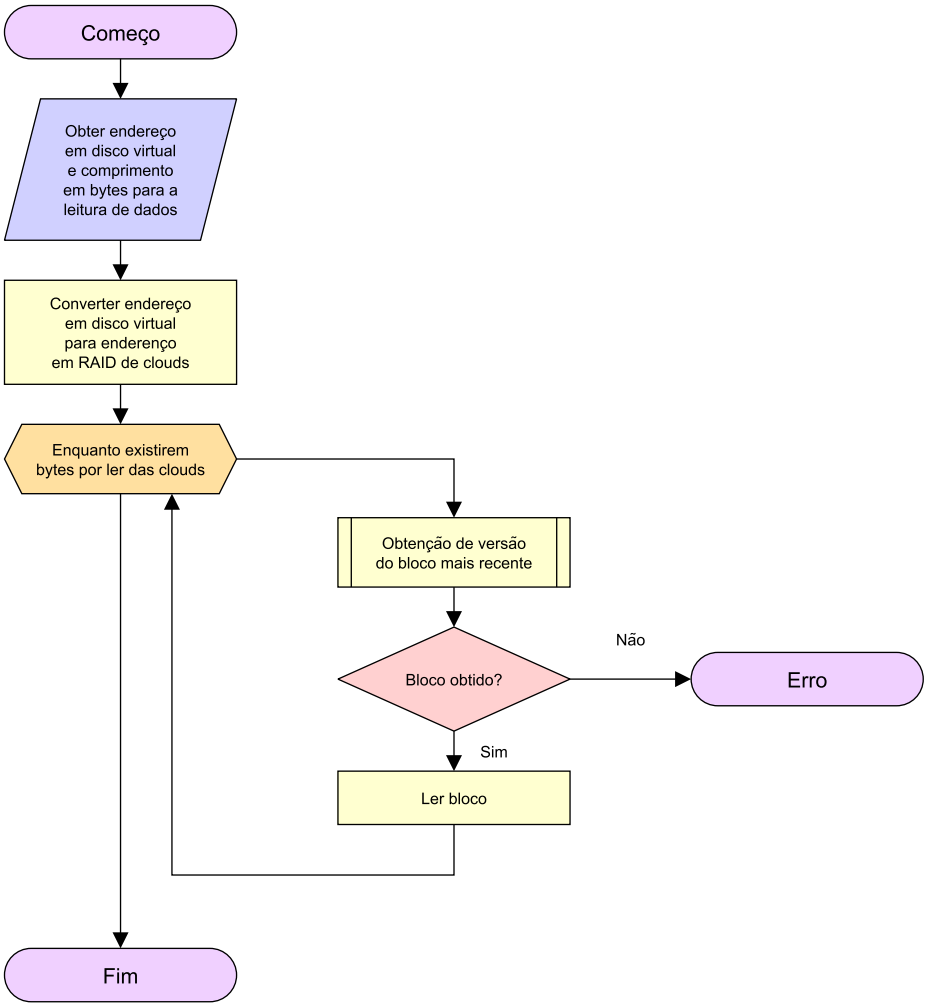


Figura 4.5: Função de Leitura do NBD

Visto que o servidor NBD implementado não é concorrente (isto é, não arranca uma *thread* para cada pedido recebido), a primeira condição é garantida por defeito. Outro fator que contribui para isso é o facto da *worker thread* (e, por consequência, as *threads* de *upload*), não alterar blocos existentes na *cache*, o que significa que, no momento de leitura de um bloco, este garantidamente não será alterado por qualquer outro agente.

A segunda condição prende-se com o funcionamento do componente de *journaling*, que mantém temporariamente uma cópia de blocos localmente alterados. Estes blocos, designados de blocos-sombra, são descritos na secção 4.5.1. Esta condição é facilmente exercida no sistema devido à forma como a numeração de blocos funciona. Sendo a numeração sequencial, começando no número 1, um bloco-sombra é identificado pelo negativo do bloco a que está associado. Para obter a versão mais recente de um bloco, basta tentar acedê-lo pelo positivo do seu número.

4.2.5 Escrita

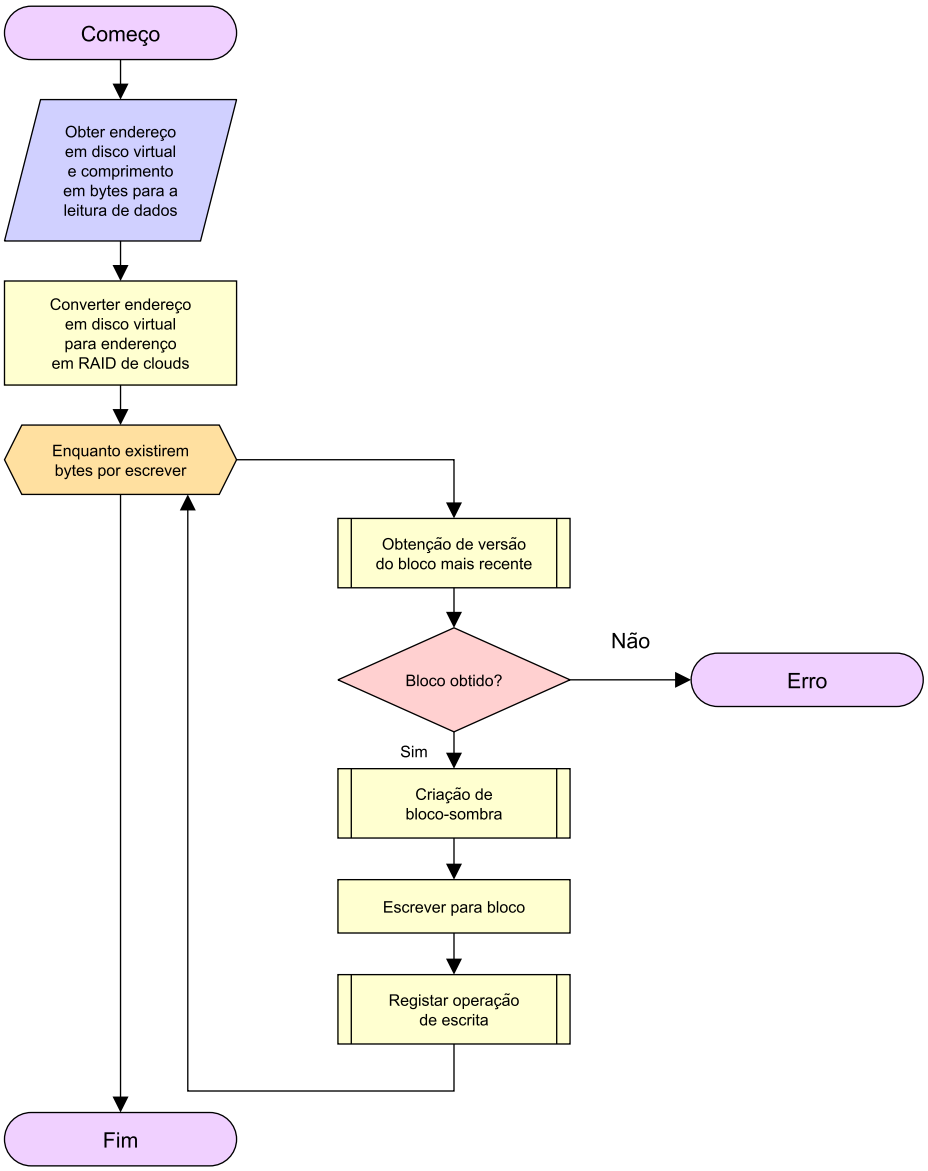


Figura 4.6: Função de Escrita do NBD

A operação de escrita recebe como argumentos o *offset* no dispositivo de blocos a partir do qual deve operar e o conteúdo a escrever. A escrita local, representada pela figura 4.6, é realizada imediatamente de forma bloqueante, mas a escrita para a *cloud* é agendada para ser realizada numa altura posterior pela *worker thread*. Tal agendamento é feito recorrendo ao componente de *journaling*, descrito na secção 4.5. Desta forma, é possível fazer uma operação penosa em termos de tempo de forma assíncrona. De notar que o cálculo de paridade não é realizado no momento de escrita local, somente durante a escrita junto das *clouds*.

Existe um problema de concorrência entre a realização desta operação e as operações a serem feitas pelas *threads* de *upload* ao mesmo tempo. Como tal, é garantido que, no momento em que uma *thread* de *upload* está a realizar uma operação de envio para uma *cloud*, a *thread* principal, encarregue desta operação de escrita, não possa escrever por cima dos blocos que a *thread* de *upload* está a ler. Ou seja, é garantido que os seguintes eventos não se verificam:

- *Thread* de *upload* está a ler bloco X. Após algumas unidades de tempo, a *thread* principal começa a escrever no mesmo bloco ao mesmo tempo que a outra ainda está a acedê-lo;
- *Thread* principal está a escrever no bloco X. Após algumas unidades de tempo, uma *thread* de *upload* começa a ler esse mesmo bloco de forma concorrente à *thread* principal;
- Ambas as *threads* acedem o mesmo bloco ao mesmo tempo;
- Em todos os casos anteriores, existe ainda o problema de a *thread* principal modificar a cópia mais recente de um dado bloco.

Esta garantia só é necessária ser exercida sobre a cópia mais recente de um dado bloco. O mecanismo utilizado para conseguir estas garantias consiste no uso de estruturas de bloqueio de acesso. Sempre que uma *thread* quiser aceder a um bloco, esta bloqueia a linha à qual o bloco pertence. Desta forma, a *thread* que conseguiu efetuar o bloqueio tem acesso exclusivo ao bloco.

Existe ainda a questão de concorrência ao acesso dos *journals*. A *thread* principal não pode escrever ao mesmo tempo que a *worker* thread está a ler os *journals*. A implementação utiliza estruturas de bloqueio de acesso para evitar isso.

4.2.6 TRIM, Flush e Desconetar

Existem ainda outras funcionalidades que podem ser implementadas num servidor de NBD, nomeadamente TRIM, *flush* e desconetar. A primeira serviria para executar a operação TRIM sobre o disco. Tal operação é disponibilizada graças ao facto dos discos NBD serem apresentados como um disco SSD. O *flush* existe para, caso o servidor NBD realize algum tipo de *buffering* temporário, o conteúdo dentro desses *buffers* seja tratado de imediato. A operação desconetar serve para o cliente assinalar que vai terminar o uso de um disco disponibilizado pelo servidor NBD, permitindo a este preparar o que achar necessário para este evento.

TRIM A operação TRIM consiste na eliminação de blocos de um disco SSD. Esta operação existe para melhorar o desempenho de escrita neste tipo de dispositivo de blocos, pois se um dado bloco onde se pretende escrever não estiver vazio, o dispositivo tem primeiro que o limpar totalmente [67]. Esta funciona limpando blocos que o sistema-operativo já sabe que não estão alocados para nada, logo o conteúdo destes já não será necessário manter. No contexto do NBD, esta operação pode ser usada por um servidor para eliminar porções específicas do dispositivo de blocos que implementa. É fácil perceber que esta função seria uma mais-valia para a implementação deste sistema, pois permitiria poupar espaço nas *clouds*. Porém, devido a falta de tempo, não foi possível desenvolvê-la completamente, tendo ficado apenas por implementações preliminares que comprovaram o seu funcionamento.

Flush A emissão de uma operação de *flush* para um servidor NBD serve apenas para sinalizar que tal operação foi pedida pelo módulo *kernel*. Isto poderia ser útil se a implementação não efetuasse todas as escritas locais imediatamente, algo que não se deverá verificar. Ou seja, o *flush* realizado pelo módulo de *kernel* tem implicações no cliente (relativamente ao *caching* efetuado pelo próprio *kernel* de blocos ou páginas), mas não necessariamente no servidor.

Desconetar Esta operação realiza-se de forma igual à de saída do programa aquando da ocorrência de erros impossíveis de recuperar durante a sua execução ou quando o utilizador envia um sinal para a morte do processo. Seria possivelmente interessante implementar esta função, mas foi determinado que o foco do desenvolvimento deveria recair sobre as funções principais.

4.3 Configuração

De forma a evitar ter que passar vários argumentos longos aquando do arranque do programa, a implementação fornece um mecanismo de leitura de ficheiros de configurações suportado pela biblioteca libconfig [68]. A sua escolha não se prende com algum fator em específico, mas a sua facilidade de uso foi um forte motivo. Esta biblioteca funciona através da leitura integral de um ficheiro de configuração com formatação própria para a memória, ficando aí residente até o objeto que contém a configuração ser destruído. Não permite acessos concorrentes à estrutura de memória, mas permite acessos a esta por múltiplas *threads*, desde que o acesso seja controlado (por estruturas de pthreads, por exemplo).

O ficheiro de configuração contém vários parâmetros personalizáveis, nomeadamente:

- Tamanho máximo de um bloco em bytes;
- Tamanho do dispositivo de blocos virtual em bytes;
- Número máximo de blocos que podem estar em cache;
- Nome do dispositivo de blocos NBD como indicado na secção 4.2.1 (`/dev/nbdX`);
- Nome do contentor a utilizar nas *clouds*;
- Pasta do sistema local utilizado para o armazenamento local da implementação;
- Prefixo para os ficheiros de *journaling*;
- Chave simétrica usada para calcular os valores MAC, devendo ter um comprimento de 32 bytes;
- Informação de acesso para cada contentor, nomeadamente Uniform Resource Locator (URL) da instância de Deltacloud (elaborado na secção 4.4), nome de utilizador e palavra-chave.

Modificação dos parâmetros, excetuando os relativos aos CSPs a aceder, deverão ser definidos na criação do dispositivo de blocos, mantendo-se fixos para o resto do tempo de vida do dispositivo.

O armazenamento de chaves e palavras-chave significa que o armazenamento deste ficheiro de configuração deve ser realizado com cuidado. A implementação não faz esforço algum para proteger o seu conteúdo, deixando essa tarefa a cargo do utilizador e do funcionamento do sistema local.

4.4 Interação com Serviços Cloud

Esta implementação recorre a uma ferramenta de abstração de interação com os serviços *cloud*. Desta forma, torna-se possível ter que produzir código específico para cada CSP utilizado. A abstração é do tipo aplicação externa, tendo sido escolhida a ferramenta Deltacloud. A razão para tal escolha deve-se às facilidades de utilização fornecidas pela ferramenta ao se utilizar a linguagem C, precisamente a mesma que foi utilizada para a elaboração desta implementação. O cenário de uso em máquina servidor motivou também esta escolha, pois permite que outras aplicações para além desta implementação possam tomar partido dela.

O Deltacloud é um programa executado à parte, isto é, não faz parte da implementação desenvolvida. Cada instância lida com comunicações para um determinado CSP. Esta recebe pedidos através de uma das três APIs que suporta através de uma ligação TCP. Tal como no caso do NBD, este não precisa de estar a ser executado na mesma máquina que a implementação. Para cada CSP diferente, é necessário existir pelo menos uma instância separada da ferramenta a correr. É possível ter mais que uma instância para o mesmo CSP em execução simultânea, de forma a paralelizar pedidos. Para o funcionamento da implementação, é criada uma instância para cada serviço de *cloud* utilizado.

Para facilitar comunicações com o programa, o projeto disponibiliza bibliotecas. A implementação recorreu a uma destas para a linguagem C, chamada libdeltacloud [21], fornecida pelos criadores do Deltacloud. A funcionalidade desta foi complementada por funções auxiliared desenvolvidas com uma outra biblioteca de nome libcurl [69]. Esta segunda fornece funções para construir pedidos HTTP próprios, permitindo desta forma interagir com o Deltacloud de forma específica.

A implementação interage com o Deltacloud na obtenção de blocos que não estão em *cache* por ambas as *threads* e em operações de escrita nas *clouds* por parte das *threads* de *upload*.

4.5 Journaling

Durante o normal funcionamento da implementação, a operação de leitura é realizada imediatamente, incluindo possível obtenção de blocos da *cloud*. No entanto, o mesmo não se verifica para as operações de escrita, mesmo tendo em conta o funcionamento *write-through* da cache. É possível deixar as operações de escrita nas *clouds* acumularem-se, de forma a reduzir o número de cálculos de paridade efetuados e não bloquear a *thread* principal com estas operações, visto que necessitam de uma quantidade considerável de tempo para serem executadas. Desta ideia surgiu a *worker thread*, responsável pela escrita nas *clouds*, juntamente com a sua ativação periódica.

Para que a acumulação de operações de escrita possa ocorrer, é necessário ter um mecanismo que registe estas operações pendentes. Este mecanismo é o chamado *journaling*. A ideia na qual este mecanismo se baseia provém de conceitos retirados de sistemas de ficheiros [31]. Para além de deixar acumular operações, este mecanismo permite, em caso de falha do sistema local, recuperar o estado de execução, ficando esta ciente do que ainda não está nas *clouds* aquando do seu arranque.

No contexto de esta implementação, o registo de operações (o *journaling* em si) é realizado sobre ficheiros chamados *journals*. Cada *cloud* utilizada tem o seu próprio *journal*, de forma a facilitar a gestão de operações pendentes. Para além de garantir que as operações de escrita nas *clouds* tomam lugar, este garante também, de forma sub-entendida, o cálculo dos blocos de paridade, pois cada operação de escrita numa *cloud* implica esse cálculo (como já foi referido na secção 4.2.5, a escrita local não despoleta esse cálculo).

Todos os ficheiros de *journal* são escritos de forma humanamente legível, manipulados através da biblioteca libconfig, a mesma utilizada para o ficheiro de configuração. Cada ficheiro consiste numa lista de várias entradas, cada uma correspondente a uma operação de escrita. Cada entrada é identificada com um número que lhe é único, permitindo através deste perceber a sequência temporal de operações de escrita, e contém no seu interior o número do bloco a escrever na sua *cloud* correspondente.

Existem vários tipos de *journaling*, como evidenciado em [70]. O tipo de *journaling* que é implementado é *journaling* de dados. Isto significa que são mantidos alguns dados e metadados sobre cada transação. A manutenção de dados é necessária para garantir a possibilidade de realização de operações relativas ao sistema de paridade. Os dados mantidos consistem em blocos-sombra: no momento imediatamente antes de um bloco ser alterado pela função de escrita do NBD, o bloco é copiado, sendo esta cópia o bloco-sombra. Esta operação realiza-se de forma a garantir consistência entre espaço local e valores de paridade. Este detalhe é elaborado na secção 4.5.1.

O tempo de vida de uma entrada de um *journal* começa com a sua inserção pela *thread* principal. Eventualmente, esta é lida pela *worker thread* no momento em que está a recolher operações pendentes. Se a operação a que uma entrada se refere for realizada corretamente junto da *cloud* correspondente, é eliminada. O tempo de vida de um bloco-sombra está descrito na secção 4.5.1.

O acesso aos vários *journals* não pode ser concorrente, visto a biblioteca libconfig não suportar acessos concorrentes. No entanto, a implementação possui duas *threads* concorrentes a acederem aos *journals*. Tal ocorre no momento em que a *thread* principal estiver a executar uma operação de escrita e, ao mesmo tempo, a *worker thread* estiver a reunir operações por executar. São utilizados mecanismos que garantem acesso exclusivo individual a cada *journal*.

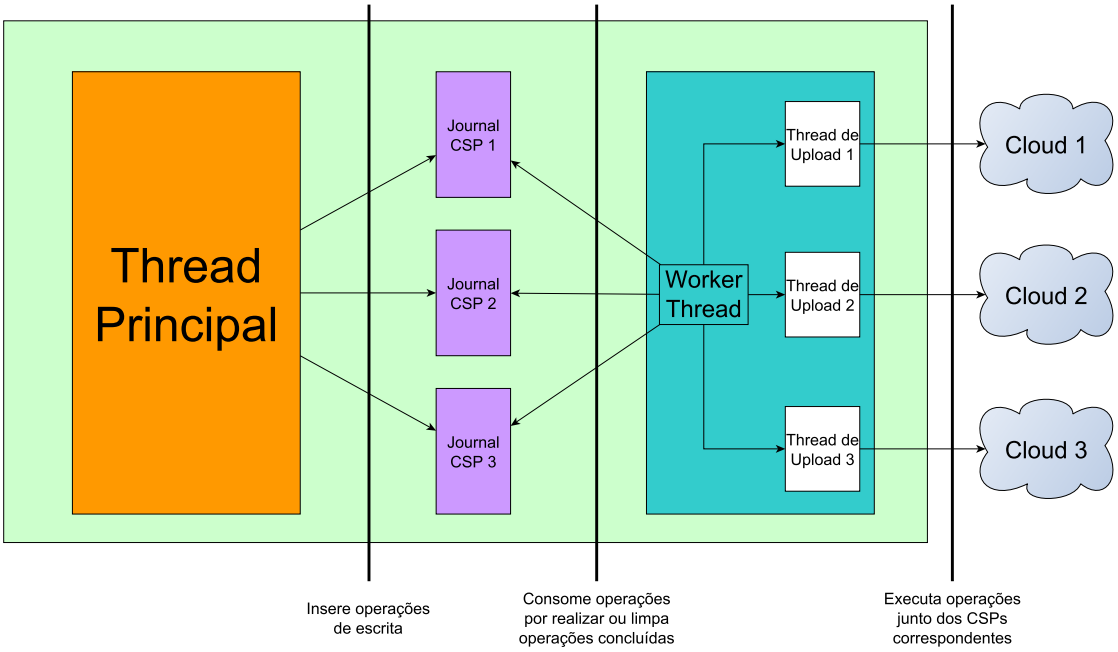


Figura 4.7: Utilização de *journals*

Relação entre *thread* principal, *worker thread* e os *journals* de cada *cloud*: *thread* principal insere operações pendentes nos *journals* e *worker thread* vai-as consumindo e eliminando, sendo a segunda operação realizada depois das transações consumidas terem sido concluídas. Neste exemplo, o utilizador contratou três *clouds*.

Como mencionado, o sistema de *journaling* permite recuperar o estado de execução em caso de falhas. Este permite ainda manter uma lista de operações pendentes de escrita nas *clouds*. O primeiro caso é lidado aquando de novo arranque da implementação, o segundo é tratado pela *worker thread* aquando da sua ativação periódica. Um cenário de falha é detetado pela implementação quando uma entrada no *journal* é criada, mas não é eventualmente removida. Isto pode-se dever a dois casos de falha:

- **Falha 1:** A implementação ou o próprio sistema local entra em estado de erro e termina abruptamente;
- **Falha 2:** A operação de escrita na *cloud* correspondente não foi realizada devido a erro de comunicação;

Para lidar com os casos de falha, a implementação simplesmente efetua as operações presentes no *journal* novamente, mantendo-as nestes até conseguir realizá-las.

4.5.1 Dados e Metadados

Como foi já mencionado, o tipo de *journaling* efetuado pelo sistema envolve o armazenamento de metadados (n.º de identificação da operação e n.º do bloco da operação) e de dados. No que toca aos dados, são guardados localmente na cache blocos-sombra.

Estes blocos-sombra consistem numa cópia de um bloco de dados exatamente antes desse mesmo bloco ser alterado. Estes são identificados pelo negativo do número do bloco de dados a que estão associados. A existência destes blocos deve-se à forma como a atualização do bloco de paridade é realizada: como esta operação não é efetuada no momento em que a escrita de um bloco de dados ocorre localmente na *cache*, então isso significa que o estado atual do bloco de paridade não tem em consideração a nova versão do bloco de dados modificado.

Se for necessário, a certa altura, realizar uma operação de recuperação de bloco e o bloco de paridade ainda não estiver atualizado, então torna-se impossível recuperar o bloco afetado. Se bloco de paridade for $B_p = B_1 \oplus B_2 \oplus B_3$ e B_1 for modificado, então existe um intervalo de tempo no qual o B_1 usado para calcular B_p é diferente do $B_{1atualizado}$.

Isso significa que no caso de recuperação de B_2, que se procederia de acordo com $B_2 = B_p \oplus B_1 \oplus B_3$, o valor de B_2 que se obteria seria incorreto, pois o valor de B_1 não é igual ao que possuía no momento em que B_p foi calculado. É precisamente para resolver esta questão que os blocos-sombra são criados. Com estes blocos, $B_2 = B_p \oplus B_{-1} \oplus B_3$, onde B_{-1} é o bloco-sombra de B_1.

O bloco-sombra permite ainda a execução de cálculos diferenciais de paridade, um processo descrito na secção 4.7.2. Este tipo de cálculo permite poupar no número de acessos efetuados. Logo, para além de uma razão de consistência de dados, tem uma razão de desempenho.

O tempo de vida de um bloco-sombra começa quando um bloco de dados é alterado. Nesse momento, se não existir já um bloco-sombra, o bloco de dados é copiado antes das alterações se realizarem. Estes blocos só são eliminados quando for atualizado localmente o valor de paridade da linha correspondente ao bloco de dados envolvido, pois nesse momento o bloco de paridade está de acordo com as versões mais recentes dos blocos de dados que lhe pertencem.

4.6 Cache

O sistema possui uma cache para explorar localidade temporal e espacial nos acessos aos diferentes blocos do dispositivo virtual. Nela, são armazenados blocos de dados, assim como blocos-sombra. A gestão dos blocos de dados é realizada através de uma política de *caching*, enquanto a gestão dos blocos-sombra é feita à parte, em pontos específicos. A unidade é, portanto, o bloco, mas ainda não foi estabelecido como esta é representada ao certo.

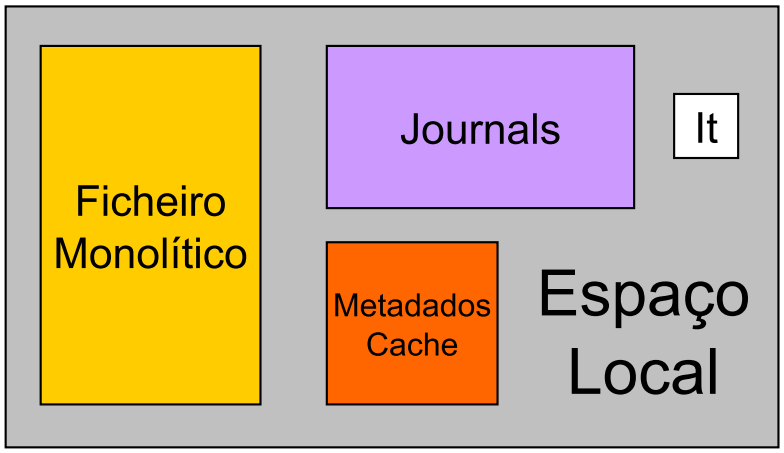


Figura 4.8: Elementos do Espaço Local

No espaço local, são armazenadas quatro tipos de coisas diferentes: um ficheiro monolítico, onde os blocos de dados (inclusive os de paridade) e blocos-sombra são mantidos; um contador para gerar o n.º de identificação das entradas dos *journals*; um número variável de *journals* igual ao número de *clouds* utilizadas; metadados da cache.

As duas principais alternativas eram guardar cada bloco como um ficheiro separado ou colocar o conteúdo dos blocos num único ficheiro monolítico particionado em casas (*slots*). Estas casas são de tamanho fixo, igual ao tamanho máximo de um bloco juntamente com o seu valor MAC. A utilização de um único ficheiro permite usar um único *file descriptor*, evitando *overhead* desnecessário que se verifica no uso de múltiplos ficheiros.

O número de casas do ficheiro é um parâmetro do ficheiro de configuração. A escolha deste número tem um impacto significativo na execução do programa. Se o número de entradas for baixo e o ficheiro ficar cheio de blocos modificados, então o programa pode terminar inesperadamente se lhe for requisitado uma operação de escrita num bloco que não está presente localmente. Nesse caso, teria de fazer *download* de novo bloco, mas não teria espaço para ele.

Blocos não são as únicas coisas a serem armazenadas localmente. Para o funcionamento pleno da cache, é necessário guardar o seu estado atual, isto é, o que está atualmente armazenado no ficheiro monolítico e em que estado está. Isto é relevante para a implementação pois, neste caso, quando um bloco é acedido, o programa verifica primeiro se ele existe localmente e, se não existir, vai à *cloud* correta obtê-lo (tal como a figura 4.4 indica). Ou seja, o espaço local toma prioridade sobre o espaço remoto. Se o espaço local estiver íntegro, então o espaço remoto eventualmente também estará. Este estado da cache é descrito por uma lista de metadados acedida através da função `mmap()`, de forma a que qualquer operação de modificação desta seja imediatamente armazenada.

Para ambos os espaços (local e remoto) se manterem completos e íntegros, é necessário saber quais blocos é que foram modificados ou não. Para saber isto, verificam-se os metadados da cache que, como foi dito, contêm o estado de cada bloco. Para além do estado, cada entrada contém o número do bloco e o seu tamanho. O estado é descrito por um bit e indica se o bloco foi alterado ou não, sendo estes os únicos dois estados possíveis. Se tiver sido alterado, então já não pode ser eliminado pela política de *caching*. O seu valor é limpo assim que o bloco em questão for escrito na *cloud*

Para além desta estrutura, a cache armazena ainda um contador para as operações do sistema de *journaling*, utilizado para gerar os n.º de identificação das operações dos *journals*, assim como os *journals* em si, cujo conteúdo é descrito na secção 4.5.

Secção 3.2.2 indica que a implementação da política recorre a duas filas diferentes. A implementação destas filas foi realizada através do uso de *hashsets*. Esta estrutura garante quatro pormenores fundamentais para o funcionamento da política: os tempos de acesso são, no pior caso, imediatados; permite acesso aleatório ao seu interior; garante que só existe uma instância de um dado valor; preserva a ordem de entrada. A implementação de *hashsets* utilizada foi a disponibilizada pela biblioteca uthash [71]. Estas filas não são guardadas localmente, pois o estado destas não afeta os dados.

4.7 RAID e Paridade

Apesar da estrutura RAID ter já sido descrita na secção 3.2.4, alguns dos seus detalhes de funcionamento ficaram por detalhar, nomeadamente quais cálculos são feitos para aceder os blocos de dados. O NBD apresenta ao cliente um dispositivo de blocos normal, o que significa que os números de bloco do dispositivo apresentado são sequenciais. Como a nossa implementação introduziu blocos de paridade, tal pormenor já não se verifica, pois estes novos blocos partilham a mesma numeração com os blocos de dados. Esta é a razão pela qual tais cálculos são necessários: sem eles, não seria possível aceder os dados corretamente.

Por detalhar ficaram também os procedimentos que se efetuam para calcular os blocos de paridade. Apesar de a forma como este é procedida ter sido explicada na secção 2.5.3, existem alguns pormenores de implementação de relevo, como atalhos e custos de leitura e escrita. Também ficou por detalhar o processo de recuperação de um bloco através de paridade.

4.7.1 Cálculos RAID

Devido à estrutura RAID 5 do dispositivo de blocos implementado, torna-se necessário efetuar cálculos para a implementação ser capaz de lidar com os diversos pedidos. É preciso também saber se os valores dados no ficheiro de configuração permitem a criação de um RAID 5 perfeito.

RAID Perfeito Para o RAID 5 ser perfeito, é necessário que o tamanho indicado para o dispositivo seja tal que, de acordo com o tamanho máximo de bloco, todas as linhas do RAID estejam preenchidas por completo. Para fazer esta verificação, realizam-se os seguintes cálculos:

$$devsize \bmod blksize == 0 \ ? \ true : \ false \tag{4.1}$$

Esta primeira equação serve para verificar se a divisão entre o tamanho indicado pelo utilizador para o tamanho do dispositivo ($devsize$) e o tamanho máximo de um bloco ($blksize$) é igual a zero. Se for, então não há blocos incompletos, isto é, blocos cujo tamanho máximo teria de ser diferente de $blksize$.

$$numblocks \bmod (ncsps - 1) == 0 \ ? \ true : \ false \tag{4.2}$$

Nesta equação, o que se está a testar é se o número de blocos de dados (ou seja, todos os blocos exceto os de paridade), indicado por *numblocks*, quando dividido pelo número de *clouds*, dado por *ncsps*, menos um é igual a zero. Se for, então todas as linhas poderão ser preenchidas por completo.

Número de Bloco e Linha Sabido que o RAID é perfeito, torna-se necessário identificar o número de um bloco e a sua linha para manipular os dados corretamente.

$$blknum = \lfloor offset/blksize \rfloor + 1 \tag{4.3}$$

A equação 4.3 serve para determinar o número de bloco. *blknum* contém o resultado, $offset$ contém o valor de *offset* a partir do qual se pretende aceder o RAID e *blksize* contém o tamanho máximo de um bloco. De notar a soma por um no final: blocos de dados têm de ter um número não-negativo (não podem ter o número zero). Com esta restrição, os blocos-sombra podem ser identificados pelo negativo do bloco que estão associados.

$$blkline = \lfloor (blknum - 1)/ncsps \rfloor + 1 \tag{4.4}$$

A equação 4.4 fornece o número de linha como resultado. *blkline* contém o resultado e *ncsps* representa o número de *clouds* a serem utilizadas. Linhas são identificadas do número um para cima por uma questão de consistência: se blocos de dados não podem ter um número negativo ou zero, então torna-se mais legível ter a mesma restrição para as linhas.

Conversão de Offset As operações de leitura e escrita no dispositivo de blocos virtual recebem, como argumento, o número de bytes a ler e o *offset* no dispositivo em bytes. Devido à estrutura estilo RAID 5 descrita pela imagem 3.3, o valor do *offset* tem de ser alterado, de forma a ter em conta os diversos blocos de paridade que se encontram armazenados. Esta alteração segue a seguinte série de equações:

$$dbn = \lfloor offset/blksize \rfloor \tag{4.5}$$

Sendo que *dbn* (*data block number*) corresponde ao número do bloco no disco virtual se este não tivesse blocos de paridade e *blksize* (*block size*) corresponde ao tamanho máximo de um bloco.

$$line = dbn/ncsps \tag{4.6}$$

line corresponde à linha do bloco no RAID, *ncsps* (número de CSPs) corresponde ao número de *clouds*.

$$psl = line \bmod ncsps \tag{4.7}$$

psl (*parity slot line*) é o número da casa do bloco de paridade na linha a ser tratada.

$$toff = offset + blksize \times line \tag{4.8}$$

$toff$ (*temporary offset*) é um valor temporário de *offset* que contém o *offset* no RAID para bloco que se pretende aceder, mas sem considerar o bloco de paridade da linha do bloco. Ou seja, se o bloco estiver à frente do bloco de paridade, então este valor ainda está incorreto.

$$rbn = \lfloor toff/blksize \rfloor \tag{4.9}$$

rbn (*real block number*) contém o número de bloco verdadeiro do bloco dentro do RAID que se pretende aceder.

$$rbs = rbn \bmod ncsps \tag{4.10}$$

rbs (*real block slot*) contém a casa do bloco na linha a que pertence.

$$oil = rbs \geq psl \ ? \ blksize : 0 \tag{4.11}$$

oil (*offset in line*) é o valor de *offset* que se deve adicionar ao $toff$ para ter o verdadeiro *offset* do bloco. Se a casa do bloco estiver à frente da casa do bloco de paridade, então tem de se adicionar um valor igual ao tamanho máximo de um bloco a $toff$, caso contrário adiciona-se zero.

$$roff = toff + oil \tag{4.12}$$

$roff$ é o valor verdadeiro de *offset* no RAID, sendo este o resultado final.

4.7.2 Cálculo de Paridade

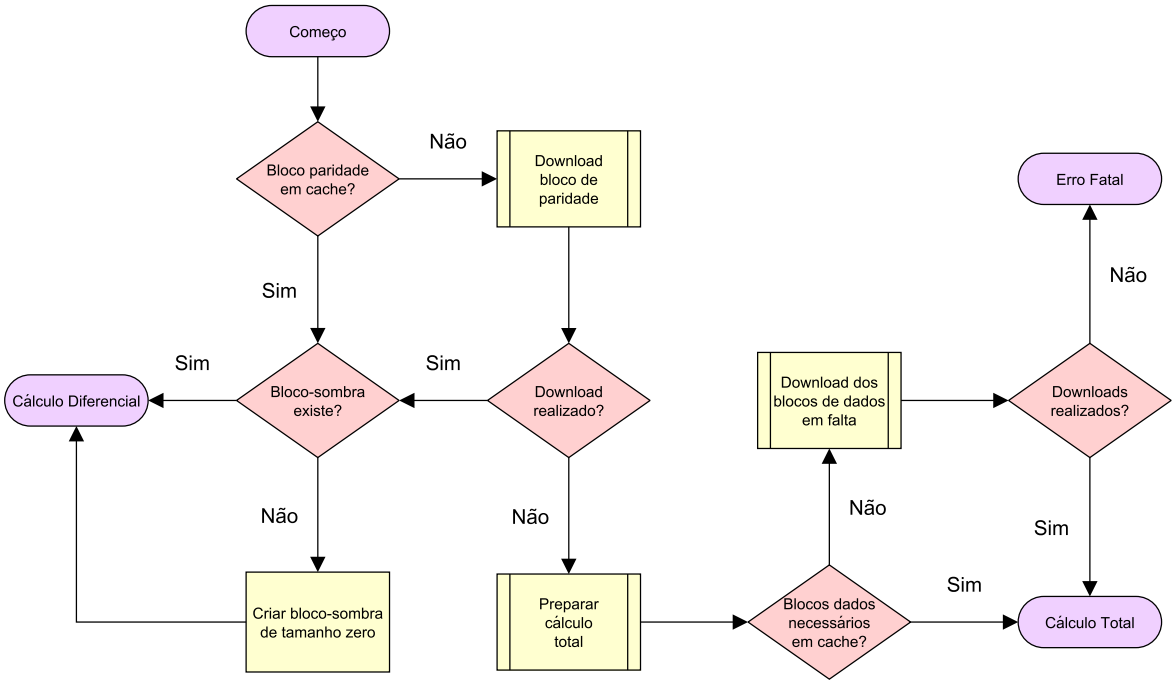


Figura 4.9: Atualização de Paridade, 1 Bloco de Dados

Assumindo que n corresponde ao número total de CSPs, x corresponde ao número do primeiro bloco de uma qualquer linha, B_x um bloco de dados e B_p um bloco de paridade:

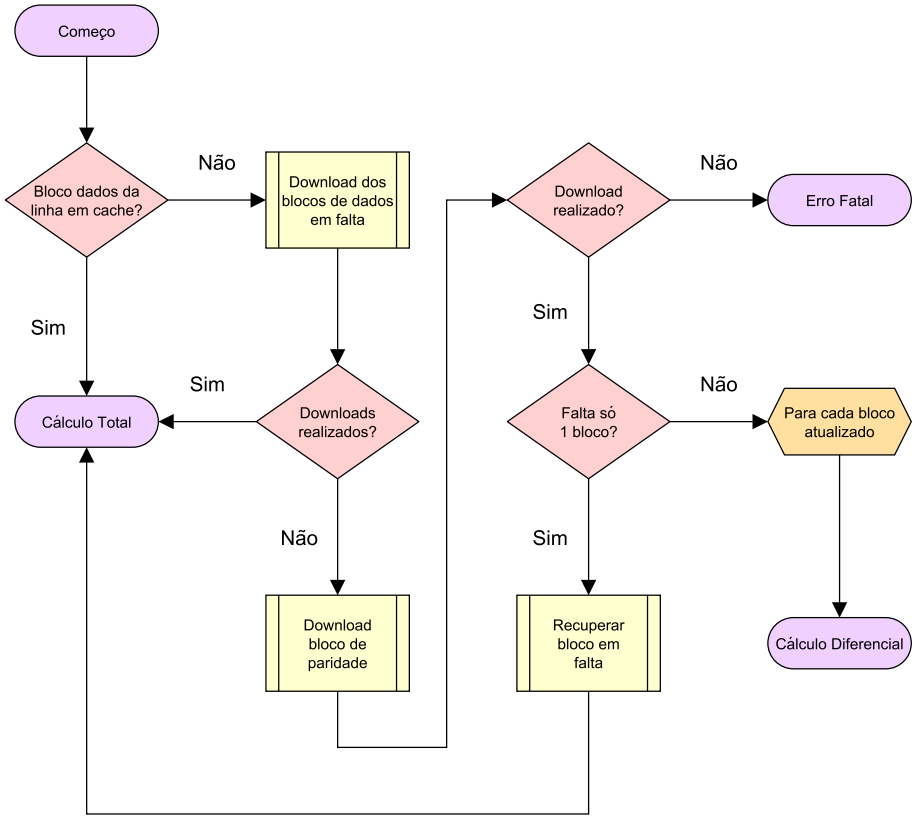


Figura 4.10: Atualização de Paridade, Mais de 1 Bloco de Dados

$$B_p = B_x \oplus B_{x+1} \oplus ... \oplus B_{x+(n-1)} \tag{4.13}$$

$$B_p = (B_{-x} \oplus B_p) \oplus B_x \tag{4.14}$$

Equação 4.13 é o chamado **cálculo total**. Para realizá-lo, é necessário todos os blocos de dados de uma dada linha estarem presente na cache local. Equação 4.14 é um **cálculo do tipo diferencial**, pois altera no bloco de paridade apenas aquilo que se pretende atualizar. De notar que B_{-x} corresponde ao bloco-sombra guardada pelo sistema de *journaling*, como descrito na secção 4.5.1.

Cálculo total é ideal quando se pretende atualizar mais que um bloco numa dada linha e obrigatório quando é necessário calcular um bloco de paridade pela primeira vez ou se tenta realizar cálculo diferencial, mas não é possível devido à incapacidade de obter o bloco de paridade. Cálculo diferencial é a melhor opção quando se quer atualizar apenas um bloco duma dada linha e a única opção quando se deseja realizar cálculo total quando uma linha possui mais que um bloco por atualizar, mas é impossível reunir localmente todos os blocos de dados dessa linha.

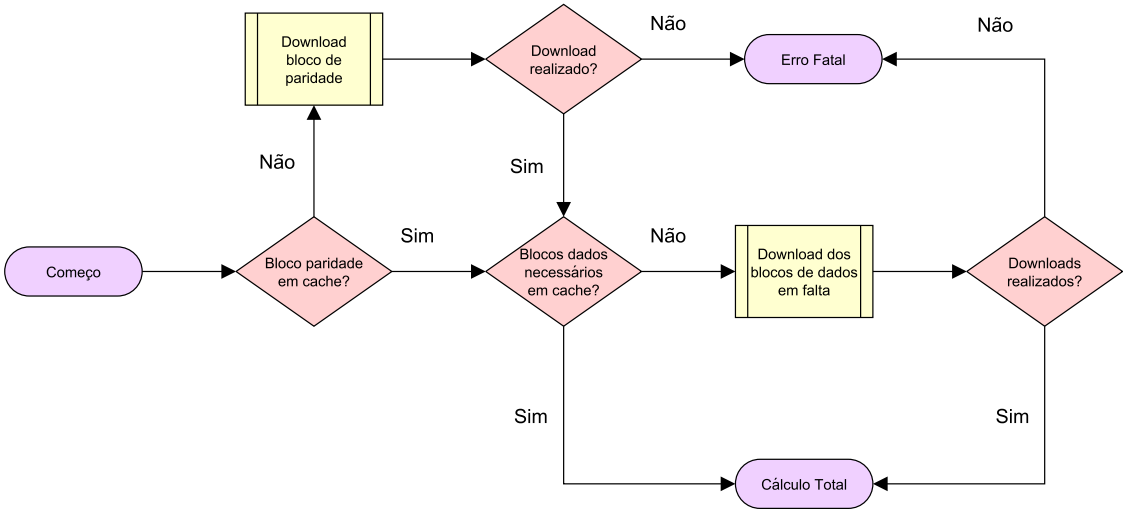


Figura 4.11: Recuperação de Bloco

A razão pela qual se deverá realizar cálculo total no momento em que é necessário atualizar mais que um bloco numa dada linha provém do número de acessos ao disco local da máquina onde a implementação estará a ser executada. Da mesma forma, quando só existe somente um bloco por atualizar numa linha, o cálculo diferencial é a melhor opção. Quantificando os acessos, novamente assumindo que n é igual ao número de CSPs:

- Cálculo total requer n acessos ao disco local, nomeadamente leitura de $n-1$ blocos de dados e escrita de 1 bloco de paridade; precisa ainda de $0 <= acessos <= n-2$ acessos remotos aos CSPs, pois pode ser necessário obter todos os blocos de dados exceto o que despoletou o cálculo em si (esse obrigatoriamente já se encontra em *cache*);
- Cálculo diferencial funciona sempre com 4 acessos locais, 3 de leitura de blocos de dado e do bloco de paridade e 1 de escrita do novo bloco de paridade; $0 <= acessos <= 1$ acessos remotos a um CSP, pois o bloco de paridade pode não estar em *cache* (o bloco que provocou o cálculo e a sua versão anterior estarão obrigatoriamente já na *cache*).

O cálculo de paridade é efetuado aquando do arranque da *worker thread*, imediatamente antes de esta começar a tratar das operações de escrita nas *clouds*. O cálculo é realizado linha por linha, ou seja, quando há mais que um bloco na mesma linha à espera de ser escrito no *journal*, em vez de se realizar a operação de cálculo de paridade múltiplas vezes, faz-se-o apenas uma vez. Como tal, a implementação reage de forma diferente aos casos em que apenas um bloco duma dada linha está por escrever ou quando são múltiplos blocos.

O processo de cálculo para quando apenas 1 bloco de uma linha é atualizado encontra-se descrito na figura 4.9; quando existem múltiplos blocos, o processo deverá ser o que se encontra elaborado na figura 4.10.

4.7.3 Recuperação de Blocos

Tal como num sistema de RAID 5, este só está preparado para recuperar da falha de apenas uma *cloud*. Se várias *clouds* falharem ao mesmo tempo, a implementação pode ou não funcionar corretamente, dependendo do conteúdo da *cache* no momento de falha. É assumido que a probabilidade de falha de múltiplas *clouds* em simultâneo é um evento pouco provável e, como tal, o uso de técnicas mais avançadas de codificação adicionaria uma complexidade considerada desnecessária a esta implementação.

O processo de recuperação está descrito na figura 4.11. Só é possível recuperar um bloco se todos os blocos de dados e o bloco de paridade duma dada linha estiverem presentes na cache (exceto, claro, o bloco em falta). Neste caso, o uso de cálculo diferencial faria pouco sentido, pois só adicionaria acessos ao disco local desnecessários. A fórmula usada para a recuperação é, então, a de cálculo total, como descrita na equação 4.13.

4.8 Segurança

A secção 3.2.3 indica, para além dos algoritmos envolvidos, como é que os elementos de segurança são obtidos. A cifragem de dados é realizada através do uso de ferramentas externas, enquanto a geração e verificação de valores MAC é lidado por uma biblioteca integrada diretamente na implementação.

A cifragem é realizada através do uso da ferramenta dm-crypt, em conjunto com o LUKS. A primeira é quem trata da cifragem em si, enquanto a segunda trata da gestão de chaves. Graças ao LUKS, torna-se possível esconder a chave-mestre usada para cifrar ou decifrar o dispositivo de blocos atrás de outras chaves. A grande vantagem desta indireção é a possibilidade de ter vários utilizadores a acederem aos conteúdos do dispositivo através de chaves personalizadas: cada utilizador tem uma chave diferente e nenhum utilizador tem a chave-mestre.

Para gerar os valores MAC, recorre-se a uma biblioteca de nome NaCl [72]. O seu funcionamento simples permitiu uma rápida integração com a implementação desenvolvida.

Capítulo 5

Discussão

Neste capítulo, são expostos elementos práticos que influenciaram as escolhas realizadas na elaboração do sistema e da implementação. Muitos destes pontos merecem exposição, pois nem todas as escolhas realizadas foram ótimas.

5.1 Linguagem de Programação

A escolha da linguagem de programação deveu-se a vários fatores. As linguagens tidas em consideração foram C e Java. Após a análise destes fatores, foi decidido que a linguagem de programação a utilizar seria C, mais especificamente C99.

Escolha das Linguagens O autor possui conhecimentos relevantes em ambas as linguagens, favorecendo Java devido à sua forte utilização no decorrer do seu percurso académico, mas sentindo-se perfeitamente capaz de programar em qualquer uma das duas. Devido ao curto espaço de tempo durante o qual foi possível realizar a implementação, foi determinado que a aprendizagem de uma linguagem que não estas duas dificultaria a execução a tempo e, como tal, mais nenhuma outra opção foi considerada.

A utilização de C99 prende-se com várias das funcionalidade introduzidas na linguagem através deste padrão: *arrays* de tamanho variável, múltiplos tipos de dados, ficheiros-cabeçalho (*headers*) novos, literais compostos e funções novas [73].

Tempo de Execução Devido às latências descritas na secção 3, foi necessário escrever o programa a produzir numa linguagem que minimizasse os tempos de execução das suas operações locais. As operações realizadas junto dos CSPs não foram tidas em conta neste critério, pois o tempo de execução delas não está dependente do recuso computacional onde o programa pode ser executado, mas sim da latência da rede, um fator que está fora do âmbito da implementação.

A linguagem C é exemplar no que toca a este ponto, aparecendo de forma consistente em primeiro lugar em *benchmarks*, nomeadamente quando compilado com o GNU Compiler Collection (GCC) [74]. Devido à falta de tempo, não foi possível realizar uma comparação direta entre as duas linguagens consideradas no nosso caso específico, pois tal implicaria uma implementação dupla do programa em questão, tendo sido determinado que os resultados de *benchmarks* generalistas seriam suficientes para uma tomada de decisão neste ponto.

Interação de Baixo Nível Devido à quantidade elevada de operações de entrada e saída de dados em armazenamento não-volátil e tendo em conta o facto de que tais operações seriam realizadas de forma paralela entre diversas *threads*, tornou-se clara a necessidade de ter um controlo muito específico de como tais operações deveriam ser realizadas. A linguagem C fornece múltiplas funções (especificadas em `unistd.h`) que são capazes de cobrir múltiplos casos de uso. Java fornece também uma variada escolha de opções, tendo mesmo um pacote (`java.nio`) dedicado a realizá-las de forma nativa ao sistema-operativo onde a máquina virtual está a ser executada. Apesar disso, o custo pago pela máquina virtual para executar estas funções é significativo, como descrito em [75].

Visto que em termos de funcionalidades ambas as linguagens são equiparáveis, foi determinado que o uso duma linguagem que funcionasse através de um agente intermediário seria desnecessário, sendo este intermediário no caso de Java a sua máquina-virtual. A utilização de tal intermediário fornece várias vantagens, nomeadamente portabilidade entre diferentes sistemas-operativos e segurança local adicional para execução de código [76]. O primeiro fator não se encaixa com o cenário de uso descrito na secção 3.1; o segundo é indiferente, pois segurança local não faz parte do âmbito do sistema implementado. Isto significa que as peculiaridades de ambas as linguagens nos dois pontos mencionados se tornam irrelevantes.

Devido à execução intensiva de operações de entrada e saída, a gestão de memória torna-se num problema muito relevante. Em C, esta gestão tem de ser feita cuidadosamente à mão, sendo necessário aquando da elaboração do programa prestar particular atenção à alocação e libertação

deste recurso. Java, equipado com os seus mecanismos de recolha de lixo (*garbage collection*) [76], torna este aspeto um não-problema, automatizando tal gestão. Isto permite o desenvolvimento de programas mais estáveis e seguros. A gestão de memória é um aspeto relevante para a implementação, o que significa que, neste aspeto, Java seria uma escolha favorável.

Para além dos fatores já mencionado, foi necessário ter em conta também que as bibliotecas usadas para o desenvolvimento da aplicação utilizam estruturas de dados próprias do sistema operativo GNU/Linux. Isto implicaria a reimplementação destas na linguagem Java, algo que foi determinado como potencialmente perigoso. Como tal, C apresenta-se neste ponto como relevante.

Bibliotecas Disponíveis Muitas das bibliotecas utilizadas na implementação possuem equivalentes em Java: libdeltacloud e JClouds para interação abstrata com *clouds*; NaCl e javax.crypto para o cálculo de valores de MAC; libconfig e pacotes nativos de Java para gestão de ficheiros de configuração; uthash [71] e pacotes nativos de Java para estruturas de dados. Visto que ambas as linguagens apresentam o mesmo conjunto de funcionalidades disponíveis através de bibliotecas (C) ou pacotes (Java), este não foi um ponto muito relevante para a determinação da linguagem a usar.

Plano de Recurso No momento em que a decisão de qual linguagem a utilizar foi tomada, não era certo o quão viável seria implementar um dispositivo de blocos virtual em espaço de utilizador. Se esta opção se revelasse como impossível de concretizar, seria necessário recorrer à opção de efetuar a implementação a nível de *kernel*. Para tal, a linguagem Java revela-se irrelevante, sendo a única opção válida a utilização da linguagem C.

5.2 Sistema de Ficheiros/Dispositivo de Blocos

Na fase inicial do desenvolvimento da implementação, a intenção era, na verdade, criar um sistema similar ao descrito neste documento, mas suportado na tecnologia FUSE. Isto deveu-se a um conjunto de suposições que, com o avançar da pesquisa sobre o estado da arte e tecnologias relacionadas, foram corrigidas.

A primeira suposição feita foi de que a única forma de desenvolver uma implementação com as características que se desejavam em espaço de utilizador seria utilizando o FUSE. Esta suposição

foi *a posteriori* dissipada, visto que a existência do NBD, juntamente com o Block Device in User Space (BUSE) [77] a servir de amostra das suas capacidades, desmentiram isto mesmo.

Uma segunda suposição que prevaleceu mesmo após a anterior ter sido desmentida foi a de que a implementação seria mais fácil de realizar recorrendo ao FUSE, ou seja, que implementar um sistema de ficheiros funcional seria mais fácil do que a criação de um dispositivo de blocos. Para além de não só se ter relativizado a dificuldade do processo de criação de um sistema de ficheiros com base numa pesquisa que, na altura, ainda se revelava incompleta, adotou-se uma posição nesta questão relativamente inflexível. O custo em termos de tempo foi bastante significativo, mas após a realização de um documento de especificação deste tal suposto sistema de ficheiros, chegou-se à conclusão que tal tarefa seria demasiado complexa e demorada para o espaço de tempo ainda disponível na altura.

Como mencionado na secção 3.2.1, a implementação de um sistema de ficheiros implica a criação de um grande conjunto de elementos: estruturas de dados, funções de manipulação de dados, organização eficiente das estruturas e distribuição de dados e metadados no disco [31]. Foi determinado não só que a complexidade da tarefa de executar estes elementos de forma correta era demasiado elevada, mas também que o nível de abstração inerente a um sistema de ficheiros impediria a implementação do sistema de redundância de dados pretendido.

5.3 Estruturas de Dados

Ao contrário de outras linguagens, C não fornece um conjunto variado de estruturas de dados pré-programadas. Como tal, torna-se necessário recorrer a estruturas fornecidas pelo sistema-operativo alvo ou à utilização de bibliotecas externas. No caso desta implementação, as estruturas utilizadas foram fornecidas pela biblioteca uthash [71]. Esta proporciona *hashmaps*, *hashsets* e listas ligadas de forma simples e eficaz.

Existem várias alternativas a esta biblioteca, nomeadamente gdsl [78], Glib [79] e Gnulib [80]. Estas proporcionam uma variedade elevada de estruturas de dados, mas a utilização destas revelou-se, no decorrer da implementação, desnecessariamente complexo. O elevado número de estruturas disponibilizadas também foi determinado como um fator negativo, pois a implementação usa somente as três estruturas supramencionadas, o que significa que fazer *linking* a bibliotecas tão pesadas seria desnecessário e de evitar.

Ao contrário das alternativas, uthash não é uma "verdadeira" biblioteca, no sentido em que é constituída apenas por ficheiros-cabeçalho. A licença desta é também diferente da das restantes, visto ser Massachusetts Institute of Technology (MIT) License em vez de GNU Public License (GPL) (ou derivada). A licença não teve impacto na escolha, mas não deixa de ser um fator que mereça menção.

A estabilidade de todas as escolhas aqui mencionadas também não foi um fator tido em conta, visto que todas elas possuem um longo ciclo de desenvolvimento, com vários anos de maturidade.

5.4 Escolha do Deltacloud

Perante a escolha da linguagem C como a linguagem de programação a utilizar na implementação e tendo em conta as alternativas listadas na secção 2.2, a escolha de uma ferramenta de abstração de acesso às *clouds* ficou reduzida a somente uma, o Deltacloud. Para utilizar esta ferramenta, a secção 4.4 indica que a implementação recorreu a uma biblioteca dos criadores do próprio Deltacloud chamada libdeltacloud.

A escolha do Deltacloud e, por consequência, da libdeltacloud não foi feita de ânimo leve. O projeto encontra-se atualmente num estado de estagnação preocupante, levando a crer que o desenvolvimento destes dois elementos está parado. Para além disso, o libdeltacloud apresenta-se como um pouco inflexível nas opções que fornece para fazer *download* e *upload* de dados. Se estes dados não constituírem um ficheiro no sistema local, então a biblioteca não permite este tipo de interação. Para resolver esta questão, recorreu-se à biblioteca libcurl para a implementação de novas funções..

Para além destes dois fatores, ainda existe a questão da ineficiência que a utilização do Deltacloud implicou para a implementação realizada. A secção 4.1.1 deixa claro que para um utilizador interagir com uma *cloud*, o sistema necessita de realizar três ligações TCP diferentes. Mesmo que duas destas sejam realizadas no sistema local através da interface *loopback*, isto provoca atrasos desnecessários que não se verificam nas restantes alternativas de abstração de interação.

Tendo todos estes fatores em consideração, a utilização do Deltacloud foi necessária, mas cara e possivelmente com um futuro incerto.

5.5 MAC e Bibliotecas

O cálculo de valores MAC é essencial para a resolução de manutenção de integridade dos ficheiros colocados nos CSPs. Como foi já mencionado, a escrita de código que seja capaz de produzir tais valores de forma correta é complexa [58], tornando-se necessária a utilização de bibliotecas desenvolvidas por indivíduos competentes na matéria.

Inicialmente, a biblioteca que se pretendia utilizar juntamente com a implementação era a libcrypto, do projeto OpenSSL [81]. Esta biblioteca disponibiliza uma vasta gama de algoritmos criptográficos para o cálculo de MACs, apresentando um longo ciclo de desenvolvimento. Tendo em conta a maturidade do projeto, o facto de ter sido certificado pelo NIST [82], uma autoridade internacional no que toca a questões criptográficas, e a vasta gama de opções por esta fornecidas, a sua escolha apresentava-se como desejável. No entanto, durante o decorrer da implementação, esta biblioteca provou ser um verdadeiro obstáculo. A sua utilização revelou-se complexa e contraproducente. Após múltiplas tentativas de integração desta no projeto sem sucesso, determinou-se que a melhor opção seria procurar uma alternativa.

Esta alternativa deveria fornecer algumas garantias de validade das operações que realiza e alguma maturidade de desenvolvimento. A escolha recaiu sobre a biblioteca NaCl [72]. Apesar de mais recente que libcrypto, o funcionamento desta foi já parcialmente validado pelos seus autores em [83] e o seu desenvolvimento no contexto de projetos da União Europeia garante um nível mínimo de qualidade. O seu uso traz dois grandes benefícios: é mais rápida e mais simples de se usar que libcrypto [84]. Porém, a sua validação encontra-se incompleta e, no momento de escrita deste documento, está num estado incerto de desenvolvimento. A última versão considerada estável foi lançada em 21 de fevereiro de 2011, o que dá a crer que o projeto aparenta ter estagnado.

5.6 Serialização de Dados

Existem vários elementos que são serializados de forma a se manterem consistentes através de várias execuções da implementação. Apesar de dois destes serem processados diretamente através do uso da função `mmap()`, nomeadamente a estrutura que descreve o estado da cache e o iterador do número de identificação de operações dos *journals*, tal não se verifica no caso dos *journals* em si, sendo estes guardados através da biblioteca libconfig.

Esta biblioteca apresenta duas grandes desvantagens que tornam o seu uso para esta funcionalidade indesejável. O primeiro problema prende-se com o facto de esta biblioteca obrigatoriamente manter em memória o conteúdo inteiro de cada *journal*. Num caso normal de uso, isto não deverá ser um problema muito significativo, pois o conteúdo destes será reduzido (visto que todas as operações de interação com os CSPs, num caso normal, deverão ser executadas à primeira, impedindo que se acumulem ao longo do tempo), mas se a falha de um CSP se prolongar por um elevado período de tempo, o tamanho dos *journals* irá crescer, podendo chegar a ocupar um espaço em memória significativo.

O segundo problema que indica que esta biblioteca está a ser usada incorretamente neste caso prende-se com a forma como esta serializa um *journal*. Cada vez que a operação de escrita da estrutura em memória é invocada, esta biblioteca apaga os conteúdos já existentes na cópia atualmente serializada em disco e escreve-os novamente na íntegra. Para tornar ainda pior a situação, esta operação de escrita recorre às funções Portable Operating System Interface (POSIX) de interação com ficheiros (`fopen()`, `fread()` e `fwrite()`), o que significa que a escrita em si pode não ocorrer imediatamente [85]. Certas precauções podem ser tomadas para evitar isto aquando do uso destas funções, precauções essas que, infelizmente, não são exercidas pela biblioteca em questão. Como tal, não há garantia de que os dados de um qualquer *journal* sejam imediatamente escritos após invocar a operação de escrita.

Estes dois fatores levantam a questão pertinente do porquê tal solução ter sido escolhida. Na verdade, não existe uma razão forte para justificar tal escolha. O uso desta biblioteca para este fim foi uma solução temporária desenvolvida durante a fase inicial da implementação, com o intuito de somente testar outros componentes. Infelizmente, devido a falta de tempo, não foi possível elaborar uma solução melhor para a serialização dos *journals*. Apesar de tudo, isto implica que o conteúdos dos *journals* é humanamente legível sem ser necessário recorrer a programas externos para a interpretação destes, algo que pode ser visto como um aspeto positivo.

A forma como a operação de leitura dos *journals* é efetuada pela implementação é irrelevante para esta questão, pois não é necessário obter garantias de que tal seja executado imediatamente.

5.7 Políticas de Caching

A política de *caching* adotada funciona à base de um esquema LRU através do uso de filas. No entanto, na fase inicial de planeamento da implementação, a intenção não era a escrita de somente uma política.

Diferentes políticas gerem uma cache de formas diferentes. Cada uma apresenta um foco específico no que toca a explorar localidade temporal ou espacial. Seria, então, relevante testar diferentes políticas, de forma a determinar qual delas seria a mais apropriada para o caso desta implementação.

Ao longo do processo de desenvolvimento da implementação, o foco deste esforço foi recaindo sobre outros elementos, cuja elaboração era mais urgente ou mais importante. Após a implementação da política LRU, restava pouco tempo para desenvolver mais soluções para esta componente. Como tal, e tendo em consideração a polivalência desta política, determinou-se que não seriam implementadas mais políticas.

5.8 Cifragem Externa/Bibliotecas

O uso de uma ferramenta externa para a cifragem de dados no sistema desenvolvido permitiu evitar a realização de uma implementação própria dos elementos criptográficos utilizados. Para além disso, permitiu o uso de uma implementação madura e analisada, algo que [58] indica como desejável. Estes elementos criptográficos encontram-se implementados não só em ferramentas externas como também em bibliotecas programáticas, logo é necessário expor as razões pelas quais se optou pela primeira opção em vez da segunda.

O cálculo de valores MAC é algo claramente programático: dá-se um *input*, operações matemáticas são realizadas e um resultado é fornecido. Estes valores são utilizados em várias ocasiões pela lógica da implementação desenvolvida durante a sua execução. Considerando a sua integração direta e profunda na implementação, não faria muito sentido recorrer a uma ferramenta externa, isto é, um executável fora do contexto da implementação, para realizar este processo.

A cifragem dos dados é um problema radicalmente diferente do anterior. Para começar, o sistema, sendo um dispositivo de blocos virtual, é cego no que respeita aos dados com que lida, pois para este é tudo um conjunto de bytes. Por conseguinte, o sistema não tem de estar ciente do estado atual dos dados. Sendo assim, não se obteria vantagem alguma em forçar a implementação a tomar consciência do processo de cifragem.

Há ainda a questão da gestão das chaves. A utilização da ferramenta dm-crypt juntamente com o LUKS permite ter múltiplos utilizadores a recorrerem ao mesmo dispositivo de blocos, cada um usando uma chave própria para decifrar os dados do dispositivo. Para resolver esta questão sem recorrer à ferramenta, seria necessário desenvolver ainda mais elementos no sistema. Outro pormenor relevante é a integração profunda de sistemas criptográficos nos sistemas-operativos, a mesma integração que torna o uso destes fácil e conveniente para o utilizador.

Enquanto o uso de uma biblioteca para o cálculo de valores MAC será justificável devido à utilização destes valores na lógica da implementação, o mesmo não se verifica para a cifragem dos dados.

5.9 Bloqueio de Linha ou Bloco

Numa fase inicial de criação da implementação, após se ter determinado a disposição dos blocos num formato igual ao de um sistema RAID 5 e elaborado a estrutura em *threads* da implementação, tornou-se claro que, para o sistema manter a validade dos blocos de paridade e a integridade do que era armazenado nas *clouds*, seria necessário implementar um sistema de bloqueio de acesso a blocos. Tornar o acesso exclusivo a um bloco para uma *thread* permitiria evitar muitas situações complicadas.

Uma destas situações ocorre durante o cálculo do bloco de paridade. Este é realizado pela *worker thread*, e esta *thread* corre ao mesmo tempo que a *thread* principal. Isto significa que, durante este cálculo, se não se recorresse a um mecanismo de bloqueio de acesso a blocos, seria possível que um dos blocos de dados envolvidos fosse modificado pela *thread* principal. Se tal ocorresse, isso tornaria o valor do bloco de paridade inválido, pois este poderia estar parcialmente dependente da versão antiga e da versão nova do bloco de dados em simultâneo.

Estabelecida esta necessidade, procedeu-se à elaboração do mecanismo com uma granularidade ao nível de blocos individuais. Este nível de granularidade apresentou-se como um passo lógico no momento de criação, mas foi-se lentamente revelando como insuficiente para fazer face aos desafios que foram surgindo.

Durante um processo de leitura de uma sequência de blocos de dados, o programa teria que libertar uma entrada na cache para realizar uma qualquer operação. Ao proceder com um bloqueio individual e sequencial de blocos, seria possível que um dos blocos a serem lidos no futuro imediato fosse entretanto eliminado pela política de *caching*.

Para fazer frente a este possível cenário, pode-se pré-calcular quais blocos se vão aceder e bloqueá-los todos preventivamente. Aparenta resolver o problema, mas esta solução tem um problema, exemplificado pelo seguinte caso: pretende-se escrever um conjunto de blocos cujo conteúdo é apenas zeros. Para bloquear o bloco, é necessário primeiro obtê-lo, o que significa efetuar uma operação de acesso a uma *cloud*. Se for um número elevado de blocos nesta situação, isto significa um atraso significativo.

A solução para este desafio foi encontrar um ponto médio entre bloqueio individual e múltiplo. Esse ponto meio, no caso da implementação do sistema, foi bloquear linhas do RAID. Esta solução é especialmente útil pois, desta forma, bloqueia-se a unidade básica do cálculo de paridade: para efetuar um cálculo do tipo total, é necessário bloquear todos os blocos de uma linha

5.10 Temporizador/Sinalização da Worker Thread

A ativação da *worker thread* é realizada após um certo intervalo de tempo. Aquando do desenvolvimento da implementação, uma das alternativas consideradas a este esquema foi o uso de técnicas de sinalização, facilitadas pela biblioteca pthreads. Quando uma nova tarefa fosse colocada num *journal*, a *thread* principal sinalizaria a realização da operação. A *worker thread*, que estaria adormecida, ativar-se-ia ao receber este sinal e processaria de imediato a operação nova.

Isto significaria que, no melhor dos casos, as operações seria lidadas quase de forma singular, isto é, cada operação de escrita seria lidada de forma separada. Tal teria implicações de desempenho, como demonstrado pela seguinte situação: realiza-se uma escrita sequencial numa linha do RAID. A *worker thread* seria notificada imediatamente após a escrita local do primeiro

bloco da linha. Realizaria a atualização do bloco de paridade dessa linha e trataria do *upload*. No entanto, faltar-lhe-ia tratar dos restantes blocos da linha. Seria obrigado a novamente calcular o valor de paridade e escrever nas *clouds*.

Para evitar isto, foi implementado o mecanismo de temporizador. Desta forma, a implementação deixa acumular um conjunto de operações. minimizando, até certo ponto, as operações de escrita nas *clouds* e de cálculo dos blocos de paridade.

Capítulo 6

Conclusões

No presente documento, foi descrito um sistema capaz de fornecer segurança e disponibilidade de dados contidos em serviços *cloud* de armazenamento. A segurança consiste na garantia de autenticidade, confidencialidade e integridade, sendo estes elementos obtidos através do uso de técnicas criptográficas. A garantia de disponibilidade é fornecida através de técnicas de codificação de dados. O sistema consiste num serviço de armazenamento de *cloud* híbrido, sendo apresentado ao seu utilizador como um dispositivo de blocos virtual que utiliza armazenamento local como *cache*, juntamente com armazenamento remoto nos serviços *cloud* para armazenamento permanente dos dados.

Confidencialidade é obtida através do uso do algoritmo de cifragem AES, também conhecido como cifra Rijndael. Como o uso da cifra em si não bastava para tratar grandes volumes de informação, recorreu-se ao modo de operação XTS, especialmente desenvolvido para o problema de cifragem de dados em dispositivos de blocos. Recorrendo a este algoritmo, juntamente com este modo de operação, um agente mal-intencionado presente num CSP não é capaz de ler os dados armazenados e qualquer alteração que realize nestes resultará numa destruição dos dados.

Como a combinação AES-XTS não era capaz de providenciar capacidades seguras de garantia de integridade de dados, tornou-se necessário utilizar um algoritmo criptográfico em separado para tal, cujo funcionamento fosse à base da produção de valores MAC. Para o sistema, foi escolhido o algoritmo SHA 2, nomeadamente a variante SHA512/256. Graças à utilização deste, tornou-se possível comparar o estado de um conjunto de dados em pontos de tempo diferentes, permitindo, assim, verificar se o conjunto foi ou não alterado por agentes mal-intencionados no lado do CSP.

A questão da disponibilidade foi resolvida através da aplicação de um esquema RAID 5, sistema este que recorre ao cálculo de valores de paridade para garantir a acessibilidade dos dados armazenados mesmo durante a quebra de fornecimento de serviços de uma das *clouds*. O uso deste esquema permite ainda a recuperação de blocos cuja a integridade foi posta em causa.

O funcionamento da implementação foi testado nos casos de modificação indevida de um bloco no espaço de armazenamento *cloud* e de indisponibilidade de uma das *clouds*, tendo funcionado corretamente, o que demonstra que o sistema é capaz de realizar as funções de redundância e de integridade corretamente. Para além disso, o conteúdo de cada bloco é trivialmente verificável como estando cifrado, o que também comprova a manutenção de confidencialidade dos dados armazenados.

Como trabalho futuro, impõe-se a implementação da função TRIM do servidor NBD. A sua utilização seria capaz de produzir poupanças no espaço gasto muito significativas. Outro elemento que merece ser elaborado em trabalho futuro é o sistema de *journaling*. A sua forma ineficaz de serialização de dados é um problema sério que merece uma melhor solução que a utilizada.

Bibliografia

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4): 50–58, 2010.

[2] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf. *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology*. National Institute of Standards and Technology, 2011.

[3] K. Ghaffari, M. Soltani Delgosha, and N. Abdolvand. Towards cloud computing: A swot analysis on its adoption in smes. *ArXiv e-prints*, 2014.

[4] B.R. Kandukuri, V.R. Paturi, and A Rakshit. Cloud security issues. In *Services Computing, 2009. SCC '09. IEEE International Conference on*, pages 517–520, Sept 2009. doi:10.1109/SCC.2009.84.

[5] M. Arregoces and M. Portolani. *Data Center Fundamentals*. Fundamentals. Pearson Education, 2003. ISBN 9781587140747.

[6] B.P. Rimal, Eunmi Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, Aug 2009. doi:10.1109/NCM.2009.218.

[7] R. Potharaju and N. Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 15:1–15:17, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi:10.1145/2523616.2523638.

[8] Gartner, Inc. Gartner: Seven cloud-computing security risks | security central - infoworld. http://www.infoworld.com/d/security-central/gartner-seven-cloud-computing-security-risks-853, 2008. (Visitado em 11/09/2014).

[9] Amazon Web Services, Inc. Amazon s3 pricing. https://aws.amazon.com/s3/pricing/, 2014. (Visitado em 15/09/2014).

[10] Microsoft Corporation. Pricing details - storage | microsoft azure. http://azure.microsoft.com/en-us/pricing/details/storage/, 2014. (Visitado em 15/09/2014).

[11] Amazon Web Services, Inc. Amazon s3 basics - amazon simple storage service. http://docs.aws.amazon.com/AmazonS3/latest/gsg/AmazonS3Basics.html, 2006. (Visitado em 15/09/2014).

[12] Microsoft Corporation. Introduction to storage | microsoft azure. http://azure.microsoft.com/en-us/documentation/articles/storage-introduction/, 2014. (Visitado em 15/09/2014).

[13] Cloud Storage Initiative. Cloud data management interface. Technical Report 1.0.2, Storage Networking Industry Association, June 2012.

[14] Amazon Web Services, Inc. Object key and metadata - amazon simple storage service developer guide. http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html, 2006. (Visitado em 15/09/2014).

[15] Cloud Storage Initiative. Cdmi server implementations | storage networking industry association. http://www.snia.org/technology-communities/cloud-storage-initiative/snia-cloud-technology-community/list-cdmi-server-imp, 2014. (Visitado em 15/09/2014).

[16] M. Kavis. Forget price. the real cloud war is about features. http://www.forbes.com/sites/mikekavis/2014/05/28/forget-price-the-real-cloud-war-is-about-features/, Maio 2014. (Visitado em 15/09/2014).

[17] N. Sadashiv and S.M.D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *Computer Science Education (ICCSE), 2011 6th International Conference on*, pages 477–482, Aug 2011. doi:10.1109/ICCSE.2011.6028683.

[18] Amazon Web Services, Inc. Overview of managing access - amazon simple storage service developer guide. http://docs.aws.amazon.com/AmazonS3/latest/dev/access-control-overview.html, 2006. (Visitado em 15/09/2014).

[19] Amazon Web Services, Inc. Protecting data using encryption - amazon simple storage service. http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingEncryption.html, 2006. (Visitado em 16/09/2014).

[20] Z. Hill and M. Humphrey. Csal: A cloud storage abstraction layer to enable portable cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 504–511, Novembro 2010. doi:10.1109/CloudCom.2010.88.

[21] The Apache Software Foundation. About. http://deltacloud.apache.org/about.html, 2011. (Visitado em 16/09/2014).

[22] enStratus Networks LLC. greese/dasein-cloud . github. https://github.com/greese/dasein-cloud, 2014. (Visitado em 16/09/2014).

[23] The Apache Software Foundation. Apache jclouds :: Home. http://jclouds.apache.org/, 2014. (Visitado em 16/09/2014).

[24] Cloudloop, Inc. Cloudloop - project kenai. https://java.net/projects/cloudloop, 2009. (Visitado em 16/09/2014).

[25] Zend Technologies Ltd. Zend framework. http://framework.zend.com/, 2014. (Visitado em 16/09/2014).

[26] W. Beary. fog - the ruby cloud services library. http://fog.io/, 2012. (Visitado em 16/09/2014).

[27] K. Perkins. pkgcloud/pkgcloud . github. https://github.com/pkgcloud/pkgcloud, 2014. (Visitado em 16/09/2014).

[28] C. Hoch. esl/elibcloud . github. https://github.com/esl/elibcloud, 2014. (Visitado em 16/09/2014).

[29] The Apache Software Foundation. Apache libcloud is a standard python library that abstracts away differences among multiple cloud provider apis | apache libcloud. http://libcloud.apache.org/index.html, 2014. (Visitado em 16/09/2014).

[30] The Apache Software Foundation. Deltacloud api. http://deltacloud.apache.org/, 2011. (Visitado em 16/09/2014).

[31] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 9a edition, 2012. ISBN 9781118063330.

[32] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi:10.1145/1774088.1774130.

[33] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[34] C. Henk and M. Szeredi. Fuse: Filesystem in userspace. http://fuse.sourceforge.net/, 2014. (Visitado em 16/09/2014).

[35] T. Nakatani. s3fs-fuse/s3fs-fuse . github. https://github.com/s3fs-fuse/s3fs-fuse, 2013. (Visitado em 16/09/2014).

[36] A. Balkan. ahmetalpbalkan/azurefs . github. https://github.com/ahmetalpbalkan/azurefs, 2014. (Visitado em 16/09/2014).

[37] P. Machek and W. Verhelst. Network block device | sourceforge.net. http://sourceforge.net/projects/nbd/, 2014. (Visitado em 17/09/2014).

[38] Sound. S3nbd. http://www.sagaforce.com/sound/s3nbd/, 2007. (Visitado em 16/09/2014).

[39] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 2a edition, 1997.

[40] P. Erdelsky. The birthday paradox. http://www.efgh.com/math/birthday.htm, Julho 2001. (Visited on 11/15/2014).

[41] V. Gough. encfs - www. http://www.arg0.net/encfs, 2010. (Visitado em 17/09/2014).

[42] T. Hicks, D. Kirkland, and M. Halcrow. ecryptfs.org. http://ecryptfs.org/, 2013. (Visitado em 17/09/2014).

[43] Cryptsetup Contributors. cryptsetup - setup virtual encryption devices under dm-crypt linux - google project hosting. https://code.google.com/p/cryptsetup/, 2013. (Visitado em 18/09/2014).

[44] TrueCrypt Developers. Truecrypt. http://truecrypt.sourceforge.net/, 2014. (Visitado em 18/09/2014).

[45] D. Goodin. Truecrypt is not secure, official sourceforge page abruptly warns | ars technica. http://arstechnica.com/security/2014/05/truecrypt-is-not-secure-official-sourceforge-page-abruptly-warns/, Maio 2014. (Visitado em 18/09/2014).

[46] Comunidade Archlinux. Disk encryption - archwiki. https://wiki.archlinux.org/index.php/Disk_encryption, Setembro 2014. (Visitado em 18/09/2014).

[47] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[48] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[49] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[50] J. Plank, J. Luo, C. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, pages 253–265, 2009.

[51] J. O'Reilly. Raid vs. erasure coding - network computing. http://www.networkcomputing.com/storage/raid-vs-erasure-coding/a/d-id/1297229, Julho 2014. (Visitado em 18/09/2014).

[52] M. Schnjakin, D. Korsch, M. Schoenberg, and C. Meinel. Implementation of a secure and reliable storage above the untrusted clouds. In *Computer Science Education (ICCSE), 2013 8th International Conference on*, pages 347–353, April 2013. doi:10.1109/ICCSE.2013.6553936.

[53] C. Henk and M. Szeredi. fuse: fuse_operations struct reference. http://fuse.sourceforge.net/doxygen/structfuse___operations.html, 2014. (Visitado em 20/09/2014).

[54] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/-Software Interface*. Newnes, 5a edition, 2013.

[55] N. Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[56] Pub, NIST FIPS. 197: Advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.

[57] J. Nechvatal, E. Barker, L. Bassham, W. Burr, and M. Dworkin. Report on the development of the advanced encryption standard (aes). Technical report, DTIC Document, 2000.

[58] B. Schneier. Schneier on security: Security pitfalls in cryptography. https://www.schneier.com/essays/archives/1998/01/security_pitfalls_in.html, 1998. (Visitado em 10/09/2014).

[59] M. J. Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. *NIST Special Publication*, 2010.

[60] IEEE 1619 Security in Storage Working Group et al. Ieee p1619/d19: Draft standard for cryptographic protection of data on block-oriented storage devices, 2007.

[61] C. Fruhwirth. *New methods in hard disk encryption*. 2005.

[62] M. J. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. *NIST Special Publication*, 2007.

[63] Pub, NIST FIPS. 180-4 secure hash standard, march 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, Março 2012.

[64] P. Hawkes, M. Paddon, and G. Rose. On corrective patterns for the sha-2 family. *IACR Cryptology ePrint Archive*, 2004:207, 2004.

[65] D. Khovratovich, C. Rechberger, and A. Savelieva. Bicliques for preimages: attacks on skein-512 and the sha-2 family. In *Fast Software Encryption*, pages 244–263. Springer, 2012.

[66] S. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. *Third-round report of the SHA-3 cryptographic hash algorithm competition*. Citeseer, 2012.

[67] J. Savill. Q. what is the trim function for solid state disks (ssds) and why is it important? | systems management content from windows it pro. http://windowsitpro.com/systems-management/q-what-trim-function-solid-state-disks-ssds-and-why-it-important, Abril 2009. (Visitado em 23/09/2014).

[68] M. A. Lindner. libconfig - c/c++ configuration file library. http://www.hyperrealm.com/libconfig/, Setembro 2012. (Visitado em 23/09/2014).

[69] Comunidade libcurl. libcurl - the multiprotocol file transfer library. http://curl.haxx.se/libcurl/, 2014. (Visitado em 23/09/2014).

[70] M. T. Jones. Anatomy of linux journaling file systems. http://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html, Junho 2008. (Visitado em 23/09/2014).

[71] T. D. Hanson. uthash user guide. http://troydhanson.github.io/uthash/userguide.html, Março 2014. (Visitado em 02/09/2014).

[72] D. J. Bernstein, T. Lange, and P. Schwabe. Introduction. http://nacl.cr.yp.to/, 2011. (Visitado em 10/09/2014).

[73] JTC 1/SC 22/WG 14. Iso/iec 9899:1999: Programming languages – c. Technical report, International Organization for Standardization, 1999.

[74] Comunidade Debian. Computer language benchmarks game. http://benchmarksgame.alioth.debian.org/, 2014. (Visitado em 24/09/2014).

[75] H. Guo and K. Xu. An in-depth examination of java i/o performance and possible tuning strategies. Technical report, University of Wisconsin, 2004.

[76] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[77] A. Cozzette. acozzette/buse . github. https://github.com/acozzette/BUSE, 2013. (Visitado em 24/09/2014).

[78] N. Darnis and P. Kerpedjiev. Gdsl - the generic data structures library, a free data structures manipulation library for c programmers. http://home.gna.org/gdsl/, 2014. (Visitado em 10/09/2014).

[79] The GNOME Project. Glib reference manual: Glib reference manual. https://developer.gnome.org/glib/stable/, 2014. (Visitado em 10/09/2014).

[80] Free Software Foundation. Gnulib - gnu portability library - gnu project - free software foundation (fsf). http://www.gnu.org/software/gnulib/, 2014. (Visitado em 10/09/2014).

[81] The OpenSSL Project. Openssl: The open source toolkit for ssl/tls. https://www.openssl.org/, 2014. (Visitado em 10/09/2014).

[82] National Institute of Standards and Technology. Validated 140-1 and 140-2 cryptographic modules. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm, 2014. (Visitado em 10/09/2014).

[83] D. J. Bernstein. Cryptography in nacl. http://cr.yp.to/highspeed/naclcrypto-20090310.pdf, 2009.

[84] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. http://cr.yp.to/highspeed/coolnacl-20120725.pdf, 2012.

[85] W.R. Stevens and S.A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional Computing Series. Pearson Education, 3rd edition, 2013. ISBN 9780321638007.